IBM ILOG Elixir V2.5

Table of contents

Getting assistance **9**

Customer Support **10**

IBM ILOG Elixir Developer’s Guide **11**

Introduction **13**

Overview 15
Introduction to IBM ILOG Elixir 17
- Aims of IBM ILOG Elixir 18
- Benefits of using IBM ILOG Elixir 19
- IBM ILOG Elixir components 20

Your IBM ILOG Elixir installation 21
The IBM ILOG Elixir License Activator 22
Building IBM ILOG Elixir applications 23
- Building an IBM ILOG Elixir application using Adobe Flex Builder 24
- Building an IBM ILOG Elixir application using the command-line compiler 26

Deploying an IBM ILOG Elixir application 27
Localizing an IBM ILOG Elixir application 28
More information 29

3D Charts **31**

Introduction to 3D charts 32
3D chart architecture 33
3D chart controls 35
- Overview of IBM ILOG Elixir 3D chart controls 36

3D column chart....37
3D bar chart....42
3D line chart....45
3D area chart....47
3D pie chart....50
3D composite chart....52
Configuring 3D chart data....54
Styling 3D charts....55
Chart styling....56
Series styling....57
Axis renderer styling....60
Managing 3D chart events....62
User interaction with 3D charts....63
Using effects in a 3D chart....64
Calendar....65
Introduction to calendars....67
Calendar architecture....68
Creating a calendar control....69
Configuring calendar data....71
Calendar data providers....72
Specifying calendar data with field mappings....73
Specifying calendar data with custom functions....75
Multiple calendars....77
Configuring a calendar control....79
Specifying the visible time range....80
Defining working and nonworking periods....84
Calendar item renderers....87
Vertical item renderer....88
Horizontal item renderer....90
Label item renderer....92
Custom item renderers....93
Styling a calendar....95
Grids....96
Column and row headers....99
Items....100
Current time indicator....101
Managing calendar events....103
Calendar data events....104
Navigation data events....105
Time range selection events....106
Sheet events....108
Event notification....109
User interaction with calendars....110

Event editing....111
About event editing....113
Defining tool tips....114
Event editing process....117
Using editing events....118
Accessing event data in an event listener....121
Determining the kind of edit in an event listener....122
Determining the reason for an itemEditEnd event....123
Examples of editing event handlers....124
Recurring events....129
Recurrence....130
Architecture of recurrence classes....131
Data fields and custom functions....132
Recurrence rules....135
Recurrence exceptions....141
Editing recurring events....145
Working with Date objects....149
Gantt Charts....151
Resource charts....153
Introduction to resource charts....155
Resource chart architecture....157
Creating a resource chart control....158
Configuring resource chart data....159
Configuring a resource chart control....165
Resource chart item renderers....179
Styling a resource chart....187
Animating a resource chart....209
Managing resource chart events....213
User interaction with resource charts....221
Editing tasks in a resource chart....227
Working with Date objects....242
Task charts....243
Introduction to task charts....245
Task chart architecture....247
Creating a task chart control....249
Configuring task chart data....250
Configuring a task chart control....257
Task chart item renderers....269
Styling a task chart....279
Animating a task chart....305
Managing task chart events....309
User interaction with task charts....319
Editing tasks in a task chart....325

Working with Date objects....339

Gauges and indicators....341

Gauges....343

Introduction to gauges....344

Gauges architecture....348

Using and customizing predefined gauges....353

Building custom gauges....365

Indicators....421

Introduction to indicators....422

Indicators architecture....423

Using indicators....424

Styling your indicators....426

Heat maps....429

Introduction to heat maps....430

Heat map architecture....432

Creating heat map controls....433

Creating a ValuedHeatMap component....434

Creating a DensityHeatMap component....436

Creating a MapHeatMap component....438

Configuring heat maps....441

Configuring heat map data....442

Configuring a color model....446

Configuring a ValuedHeatMap component....447

Configuring a DensityHeatMap component....450

Configuring a MapHeatMap component....451

Adding a heat map legend....454

Maps....457

Introduction to maps....459

Map architecture....460

Map features....461

Creating a map control....463

Styling maps....465

Overview of map customization....466

Using CSS....467

Using inline properties....468

Using tags and tag bindings....469

Using symbols....470

Using a repeater control....472

Managing map events....474

User interaction with maps....476

Using effects in a map....477

Using built-in effects in a map....479

Color transition effect....480

Fit effect....481
Using extra map features....483
Drawing....484
Longitude/latitude methods....486
Synchronizing a component with a map....488
Generating custom maps....491
Getting started....493
Getting to know the IBM® ILOG® Elixir Custom Map Converter tool....499

OLAP and pivot charts....507

Introduction to OLAP and pivot charts....508
OLAP/pivot charts architecture....511
OLAP charts....513
Creating an OLAP chart....514
Configuring an OLAP chart....518
Styling an OLAP chart....524
Pivot charts....527
Creating a pivot chart....528
Configuring a pivot chart....530
Styling a pivot chart....532
Managing PivotChart events....533

Organization Charts....535

Introduction to organization charts....537
Organization chart architecture....539
Creating an organization chart control....540
Configuring organization chart data....541
Organization chart data providers....542
Specifying organization chart data with field mappings....543
Specifying organization chart data with custom functions....546
Organization chart layout....549
Standard....550
Right hanging....551
Left hanging....552
Both hanging....553
Assistants....554
Specifying the layout algorithm....555
Organization chart view modes....557
Global view mode....558
Local view mode....559
Organization chart item renderers....561
Default item renderer....562
Detailed item renderer....563
Setting the item renderer....565
Custom item renderers....566

Styling organization charts....569
Layout properties....570
Link properties....572
Item properties....573
Managing organization chart events....575
User interaction with organization charts....576
View manipulation API....578
Adding managed data fields....579
Printing organization charts....585
Printing organization charts on multiple pages....586
Adjusting the print layout....588
Sizing a printed organization chart....590
Example of printing with multipage controls....591
Radar Charts....597
Introduction to radar charts....599
Radar chart architecture....601
Creating a radar chart control....603
Configuring radar chart data....604
Radar chart axes....605
Series in radar charts....607
Radar line series....608
Radar bubble series....610
Radar plot series....614
Radar column series....617
Radar chart axis renderers....621
Radar chart data renderers....622
Managing radar chart events....624
User interaction with radar charts....625
Using effects in a radar chart....626
Treemaps....627
Introduction to treemaps....628
Creating a treemap control....630
Configuring treemap data....631
Treemap data providers....632
Specifying treemap data with field mappings....633
Specifying treemap data with custom functions....635
Styling treemaps....636
Examples of property usage....639
Managing treemap events....642
User interaction with treemaps....644
Treemap legend control....646
Index....647

Getting assistance

Explains how to get technical support.

In this section

Customer Support
Describes the customer support available for IBM ILOG Elixir users.

Customer Support

There is an online forum and a product web site available forIBM® ILOG® IBM ILOG Elixir.

Online forum

IBM encourages you to register on the IBM® ILOG® IBM ILOG Elixir forum accessible at *http://forums.ilog.com/elixir/* to discuss with peers, provide feedback to the IBM® ILOG® IBM ILOG Elixir team and foster discussions on the future of the product.

Web sites

To obtain more information about IBM® ILOG® IBM ILOG Elixir, connect to the product Web site *http://elixir.ilog.com*.

If you require direct assistance, please contact Adobe support through the Web site *http://www.adobe.com/support/flex*.

IBM ILOG Elixir Developer’s Guide

Documents the IBM ILOG Elixir components.

In this section

Introduction
Gives general information about IBM ILOG Elixir and explains how it relates to Adobe® Flex® 3.

3D Charts
Describes the use of IBM ILOG Elixir 3D charts with Adobe® Flex® 3.

Calendar
Describes the use of IBM ILOG Elixir calendars with Adobe® Flex® 3.

Gantt Charts
Describes how to use IBM ILOG Elixir Gantt charts with Adobe® Flex® 3.

Gauges and indicators
Describes the use of IBM ILOG Elixir gauges and indicators with Adobe® Flex® 3.

Heat maps
Describes the use of IBM ILOG Elixir heat maps with Adobe® Flex® 3.

Maps
Describes the use of IBM ILOG Elixir maps with Adobe® Flex® 3.

OLAP and pivot charts
Describes the use of IBM ILOG Elixir OLAP and pivot charts with Adobe® Flex® 3.

Organization Charts
Describes the use of IBM ILOG Elixir organization charts with Adobe® Flex® 3.

Radar Charts
Describes the use of IBM ILOG Elixir radar charts with Adobe® Flex® 3.

Treemaps
Describes the use of IBM ILOG Elixir treemaps with Adobe® Flex® 3.

Introduction

Gives general information about IBM ILOG Elixir and explains how it relates to Adobe® Flex® 3.

In this section

Overview
Gives an overview of IBM ILOG Elixir and Adobe® Flex® 3 and explains the system requirements for IBM ILOG Elixir.

Introduction to IBM ILOG Elixir
Presents the aims and benefits of IBM ILOG Elixir and lists the components in this version.

Your IBM ILOG Elixir installation
Describes the installation directory contents after installation.

The IBM ILOG Elixir License Activator
Describes the purpose of the IBM ILOG Elixir license activator and where to find it.

Building IBM ILOG Elixir applications
Describes how to compile (build) your IBM ILOG Elixir application by using Adobe® Flex® Builder™ or the command-line compiler.

Deploying an IBM ILOG Elixir application
Describes how to deploy IBM ILOG Elixir applications with an RSL (Runtime Shared Library).

Localizing an IBM ILOG Elixir application
Describes how to localize an IBM ILOG Elixir application.

More information
Describes where to find more detailed information in the Adobe® Flex® documentation.

Overview

IBM ILOG Elixir overview

IBM ILOG Elixir is a set of components that extends the Adobe® Flex® component set. The IBM ILOG Elixir components can be deployed in Web-based (Flex) applications or in Adobe Integrated Runtime (AIR™) desktop-based applications. IBM ILOG Elixir provides 2D and 3D charts including pivot charts, treemaps, heatmaps, map-based dashboards, calendars, organization charts, gauges and indicators, and resource-based and task-based Gantt charts. It is designed to help you build better Rich Internet Application (RIA) and Rich Desktop Application (RDA) graphical user interfaces (GUIs) faster.

Adobe Flex 3 overview

Adobe Flex 3 is an RIA platform that includes several products:

- Adobe Flex Software Development Kit (SDK): the core platform including the ActionScript® and MXML compiler and libraries.
- Adobe Flex Builder: the Integrated Development Environment (IDE).
- Adobe® Flex® Builder(TM) Professional: contains the Adobe Flex Builder with a profiler, some data visualization components, and some 2D charts.
- Adobe Livecycle Data Services ES: an enterprise data service framework to communicate with the client and server.

For more information, see the Adobe document *Using Flex Builder 3*.

For all references to Adobe documentation, see the URL *http://www.adobe.com/go/flex3_livedocs*.

System requirements

To use IBM ILOG Elixir, you need the following software:

- Development time
    - Adobe Flex SDK 3.2
    - Adobe® Flex® Builder(TM) Professional 3.0.2 or above

      Most IBM ILOG Elixir components either inherit from Adobe Flex Builder Professional data visualization components or use them. For this reason, Adobe® Flex® Builder(TM) Professional is required to compile IBM ILOG Elixir applications. The Adobe data visualization package is only provided with Adobe® Flex® Builder(TM) Professional.
- Run time
    - Adobe Flash® Player™ 9.0.115 or higher or 10.0.12.36 or higher.

Optionally, communication with the server can be done by using Adobe Livecycle Data Services.

Introduction to IBM ILOG Elixir

Presents the aims and benefits of IBM ILOG Elixir and lists the components in this version.

In this section

Aims of IBM ILOG Elixir
Explains the aims of IBM ILOG Elixir.

Benefits of using IBM ILOG Elixir
Explains the benefits of IBM ILOG Elixir.

IBM ILOG Elixir components
Describes the IBM ILOG Elixir components.

Aims of IBM ILOG Elixir

IBM ILOG Elixir is a set of advanced visual components for the Adobe® Flex® platform, designed to enable efficient delivery of high-performance Rich Internet Applications (RIAs). RIAs take advantage of Flash Player™ 9 to combine the responsiveness and richness of desktop software with the broad reach of Web applications to deliver a more effective user experience. This technology allows developers to extend browser capabilities seamlessly, to deliver richer, more responsive client-side applications, and to achieve more robust integration with server-side functionality and service-oriented architectures.

IBM ILOG Elixir can also be used in conjunction with Adobe Integrated Runtime (AIR™) to build and deploy RIAs for use on your desktop.

Benefits of using IBM ILOG Elixir

IBM ILOG Elixir delivers many benefits, including:

♦ **Faster time to productivity**

With 20 years of graphical component development experience and a commitment to adhere to Flex® 3 standards and conventions, IBM® ILOG® has made sure that the design of IBM ILOG Elixir is an elegant yet simple component architecture that Adobe® Flex® developers will feel comfortable with straight away. To help developers to be productive faster, IBM® ILOG®has tightly integrated IBM ILOG Elixir with the Adobe Flex Builder™ IDE and delivers a rich set of documentation and samples with the software.

♦ **A rich set of functionality**

IBM ILOG Elixir is a set of components designed to cover many data display needs, such as:

- Advanced business dashboards with gauges, dials, and indicators; 3D charts; pivot and OLAP charts; heat maps; maps displays; and radar charts (also known as spider or web charts)
- Visual data mining and analysis applications leveraging OLAP and pivot charts, treemaps, and heat maps
- Planning and scheduling displays with calendars or Gantt resource and task charts to visualize planned activities and project assignments for numerous resources over long periods of time, to reassign tasks, and to manipulate time series data and schedules
- Advanced employee database displays, with organization charts to navigate quickly through peer and management relationships

Overall, IBM ILOG Elixir comes packed with the components you need to deliver rich user experiences, through advanced animation and rich programmatic capabilities.

IBM ILOG Elixir has been designed for the Adobe Flex 3 platform from the ground up and delivers:

♦ **Transparent deployment**

Web applications execute on Flash® Player™ 9, which is widely deployed on most end-user computers. This provides solid cross-browser compatibility to deliver the same experience whatever the target platform. All you need to do is update your application on the Web servers and all users get instant access to the latest fixes and enhancements.

♦ **Easier application development and maintenance**

IBM ILOG Elixir builds on the Adobe Flex 3 productive language and development environment. ActionScript® and MXML are designed to existing standards. MXML is XML compliant, implements styles based on the Cascading Style Sheets level 1 (CSS1) specification, and implements an event model based on a subset of the W3C DOM Level 3 Events specification. ActionScript is an ECMAScript-based language that provides support for object-oriented development. The classes and functionality specified in the ECMAScript for XML specification are referred to as E4X.The Adobe Flex server code executes on standard J2EE platforms or servlet containers.

IBM ILOG Elixir components

The IBM ILOG Elixir product consists of a family of related components that let you design, develop, and deploy an entirely new class of RIA. The IBM ILOG Elixir components conform to the Adobe® Flex® platform and existing Adobe Flex component paradigms. Each component can be placed at precise coordinates, sized, localized, and skinned. For details of skinning, see Adobe Flex documentation on the Flex Skin Design Extensions.

The IBM ILOG Elixir components are:

♦ Radar charts, see *Radar Charts* for more information.

♦ 3D charts, see *3D Charts* for more information.

♦ Treemaps, see *Treemaps* for more information.

♦ Organization charts, see *Organization Charts* for more information.

♦ Maps, see *Maps* for more information.

♦ Gantt charts, see *Gantt Charts* for more information.

♦ Gauges and indicators, see *Gauges and indicators* for more information.

♦ Heat maps, see *Heat maps* for more information.

♦ OLAP and pivot charts, see *OLAP and pivot charts* for more information.

♦ Calendars, see *Calendar* for more information.

Your IBM ILOG Elixir installation

Once installed, IBM ILOG Elixir is typically in one of the following directories:

- ♦ On Microsoft® Windows® the installation directory is normally: `c:\Program Files\IBM\ILOG\Elixir 2.5`
- ♦ On Mac OS® the installation directory is normally: `/Applications/IBM/ILOG/Elixir 2.5`
- ♦ On Linux® , the installation directory is normally:

  `~/IBM/ILOG/Elixir 2.5`, for example:

  `/usr/julian/IBM/ILOG/Elixir 2.5`

The installation directory contains the IBM ILOG Elixir libraries (SWC and SWF files), IBM ILOG Elixir samples, and some utilities such as the License Activator and the IBM® ILOG® Elixir Custom Map Converter.

According to the information you gave during the IBM ILOG Elixir installation process, the installation process for Eclipse® or Flex® Builder™ also installs documentation. The documentation is available through the standard Eclipse Help menu or Flex Builder Help menu.

You can run the samples from the **Start>IBM ILOG>IBM ILOG Elixir** menu (for Microsoft® Windows®), from the `<installation-dir>/samples` directory (for Mac OS®), or from the `<installation-dir>/samples` directory (for Linux®). Each sample contains its source code and a build executable that allows you to recompile the sample if necessary.

The IBM ILOG Elixir License Activator

Use the IBM ILOG Elixir License Activator to:

♦ Allow you to remove the watermark displayed by IBM ILOG Elixir components.

♦ Unpack the IBM ILOG Elixir source code.

Important: The License Activator is not available in the trial version of IBM ILOG Elixir; the only way to license IBM ILOG Elixir is to purchase and install an official version of IBM ILOG Elixir that does contain the License Activator.

You can access the License Activator through the **Start>IBM ILOG>IBM ILOG Elixir** menu (for Microsoft® Windows®), through the `<installation-dir>/LicenseActivator` directory (for Mac OS®), or through the `<installation-dir>/LicenseActivator/Activator` script (for Linux®).

Once you have entered the license key, the source code should be available in the `<installation-dir>/frameworks/projects/ilog-elixir/src` directory and the watermark no longer appears on new applications. To remove the watermark from existing applications, you must recompile them. For details, see *Building IBM ILOG Elixir applications*.

Building IBM ILOG Elixir applications

Describes how to compile (build) your IBM ILOG Elixir application by using Adobe® Flex® Builder™ or the command-line compiler.

In this section

Building an IBM ILOG Elixir application using Adobe Flex Builder
Describes how to build an IBM ILOG Elixir application using Adobe® Flex® Builder™ .

Building an IBM ILOG Elixir application using the command-line compiler
Describes how to build an IBM ILOG Elixir application using the command-line compiler.

Building an IBM ILOG Elixir application using Adobe Flex Builder

Setting up the application project

You can use the IBM ILOG Elixir components in an Adobe® Flex® application alongside Adobe Flex components.

For more information about building your application using Adobe Flex Builder, see *Building Projects* in the *Using Flex Builder 3* Adobe documentation.

To set up the application project for your IBM ILOG Elixir application:

1. In Adobe Flex Builder create a new Adobe Flex project, either for the Web (run in Flash® Player™), or for the Desktop (run in AIR™), by clicking **File>New**.
2. Click the right mouse button on your project and choose the **Properties** menu item.
3. Choose the **Flex Build Path** or **AIR Build Path** section accordingly.
4. In the library path tab, add the IBM ILOG Elixir SWC files to your project using the **Add SWC...** button.

   IBM ILOG Elixir comes in the form of two SWC files:

   ♦ The file `ilog-elixir.swc` contains the IBM ILOG Elixir classes and can be found in the `<installation-dir>/frameworks/libs` directory.

   ♦ The file `ilog-elixir_rb.swc` contains US resource bundles for IBM ILOG Elixir and can be found in the `<installation-dir>/frameworks/locale/en_US` directory.

   Note: If you are using Adobe® Flex® Builder(TM) Professional, the Adobe data visualization package (datavisualization.swc), which is required by most IBM ILOG Elixir components is automatically included. If you are using Adobe Flex Builder Standard you need to add datavisualization.swc to your library path.

5. Optionally, you can indicate to Adobe Flex Builder where it can find the IBM ILOG Elixir source code to help with debugging.

   In the `Flex Build Path` or `AIR Build Path` section:

   ♦ Expand the `ilog-elixir.swc` tree item and edit the `Source attachment` item.

   ♦ Specify the path to the source: `<installation-dir>/frameworks/projects/ilog-elixir/src`.

   To have the source code available you need first to activate IBM ILOG Elixir, see *License Activator* in *Your IBM ILOG Elixir installation*.

For details of the installation directories for Microsoft® Windows® , Mac OS® , and Linux® , see *Your IBM ILOG Elixir installation*.

When you have created your application files, you can then do one of the following:

- Instantiate them in ActionScript® files (`.as`):

  Example: `var myTreeMap:TreeMap = new TreeMap();`

- Use them in MXML files (`.mxml`):

  Example: `<ilog:TreeMap id="myTreeMap"/>`

- Use them in a design view:

  You can use the drag-and-drop feature to move components from the IBM ILOG Elixir folder of the component panel onto the design view.

Compiling and running the application

You compile and run the application from the main MXML file.

To compile and run the application:

- Click the right mouse button on the main MXML file of your application and choose the **Run Application** menu item to get your IBM ILOG Elixir application compiled and running.

Building an IBM ILOG Elixir application using the command-line compiler

For more information about compiling your application, see *Using the Flex Compilers* in the *Application Development* Adobe® documentation.

Creating the application files

To create the application:

1. Create your application ActionScript® and MXML files with your preferred editor.
2. Use IBM ILOG Elixir components by:

   ♦ Instantiating them in ActionScript files (`.as`).

   Example: `var myTreeMap:TreeMap = new TreeMap();`

   ♦ Using them in MXML files (`.mxml`).

   Example: `<ilog:TreeMap id="myTreeMap"/>`

Compiling the application on the command line

To compile the application:

1. Use the following command line for a Web-based Adobe Flex application to be run in Flash Player:

```
mxmlc -library-path+=<installation-dir>/frameworks/libs/ilog-
elixir.swc,<installation-dir>/frameworks/locale/en_US/ilog-
elixir_rb.swc myElixirApplication.mxml
```

2. Use the following command line for a Desktop-based Adobe Flex application to be run in AIR™ :

```
amxmlc -library-path+=<installation-dir>/frameworks/libs/ilog-
elixir.swc,<installation-dir>/frameworks/locale/en_US/ilog-
elixir_rb.swc myElixirApplication.mxml
```

Note: If you are using Adobe® Flex® Builder(TM) Professional mxmlc compiler, the Adobe data visualization package (datavisualization.swc), which is required by most IBM ILOG Elixir components is automatically included. If you are using Adobe Flex Builder Standard or the SDK compiler, you need to add datavisualization.swc to your library path.

Deploying an IBM ILOG Elixir application

You can deploy your IBM ILOG Elixir application just like any other Adobe® Flex® application, see *Deploying Flex Applications* in the *Application Deployment* Adobe® documentation for more information.

If your application makes use of many IBM ILOG Elixir features, use a Runtime Shared Library (RSL) to cache the entire library on the client browser instead of linking with the subset you are using.

For example, enter the following command line:

```
mxmlc -library-path+=<installation-dir>/frameworks/libs/ilog-
elixir.swc,<installation-dir>/frameworks/libs/ilog-elixir_rb.swc -
runtime-shared-libraries=ilog-elixir.swf myElixirApplication.mxml
```

Use the RSL Shock Wave File (SWF) located in the directory: `<installation-dir>/frameworks/rsls/ilog-elixir.swf`.

Alternatively, compile your application with an RSL by using Adobe Flex Builder and choosing the RSL option when you add the library to your project. For more information on RSL see *Using Runtime Shared Libraries* in the *Application Development* Adobe® documentation.

Localizing an IBM ILOG Elixir application

If you want to run your IBM ILOG Elixir application in a language other than those provided, you can localize the IBM ILOG Elixir properties by creating your own resource bundles.

To create resource bundles:

1. Activate the product to get the sources of the resource bundles.
2. Copy the directory `<installation-dir>/frameworks/projects/ilog-elixir/bundles/en_US/` to your project directory.
3. Rename the `en_US` directory you have just copied to the name of the locale you want to use, for example, `fr_FR`.
4. Translate the contents of each of the files in the new directory into the language of your choice.
5. Recompile the translated property files into a SWC using the `compc` command.
6. Include the translated bundles when you compile your application with the chosen locale, for example:

```
mxmlc -locale fr_FR -source-path+=<project-path>/{locale} -
library-path+=<installation-dir>/frameworks/libs/ilog-elixir.swc
myElixirApplication.mxml
```

When you run your application, all the IBM ILOG Elixir messages will be displayed in your chosen language. In addition, for your application to be fully localized, you must localize the Adobe® Flex® framework. For more information, see *Localizing Flex Applications* in the *Advanced Flex Programming* Adobe documentation.

More information

Once you have read the introduction to IBM ILOG Elixir and the overview of how to develop IBM ILOG Elixir applications in this guide, you may require more information about some subjects. The Adobe® Flex® documentation set is at *http://www.adobe.com/go/flex3_livedocs* and contains detailed information as follows:

♦ For information on MXML and ActionScript® , see *Developing Applications in MXML*, and *Using ActionScript* in *Flex Programming Elements*.

♦ For information on using Adobe Flex Builder™ , see *Using Flex Builder 3*.

♦ For information on using Adobe Flex components, see the *User Interfaces*.

♦ For more information on the Adobe Flex data model, see the *Using Data Providers and Collections* in *User Interfaces*.

♦ For more information on configuring, developing, and deploying Adobe Flex applications, see *Application Development* and *Application Deployment*.

♦ For information on debugging, see *Using Flex Builder 3* and the *Using the Command-Line Debugger* in *Application Development*.

3D Charts

Describes the use of IBM ILOG Elixir 3D charts with Adobe® Flex® 3.

In this section

Introduction to 3D charts
Describes the features of 3D charts with an example.

3D chart architecture
Describes the architecture of the 3D chart classes.

3D chart controls
Describes the 3D chart controls available.

Configuring 3D chart data
Describes how to configure 3D chart data.

Styling 3D charts
Describes the common properties that can be used for styling the charts, the series, and the axes in 3D charts. These are in addition to the specific properties for each 3D chart control.

Managing 3D chart events
Describes the management of events on 3D charts.

User interaction with 3D charts
Describes the ways that you can interact with a 3D chart.

Using effects in a 3D chart
Describes the use of 3D chart effects.

Introduction to 3D charts

IBM ILOG Elixir delivers 3D charts that can be displayed in either orthographic projection or oblique projection. Adjustable parameters are: `elevation`, `rotation`, `depth`, and `light direction`.

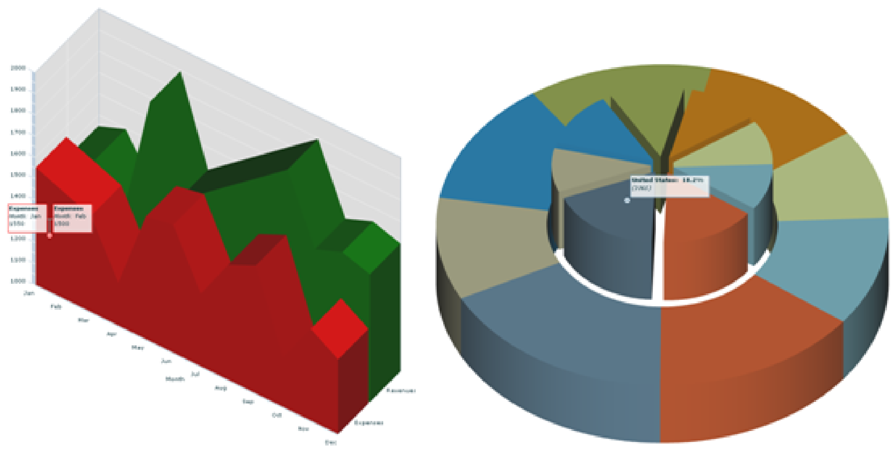
The following types of 3D chart are available:

- Column
- Bar
- Line
- Area
- Pie

IBM ILOG Elixir 3D charts are based on Adobe® Flex® Builder™ Professional and to be fully functional the following features require its installation:

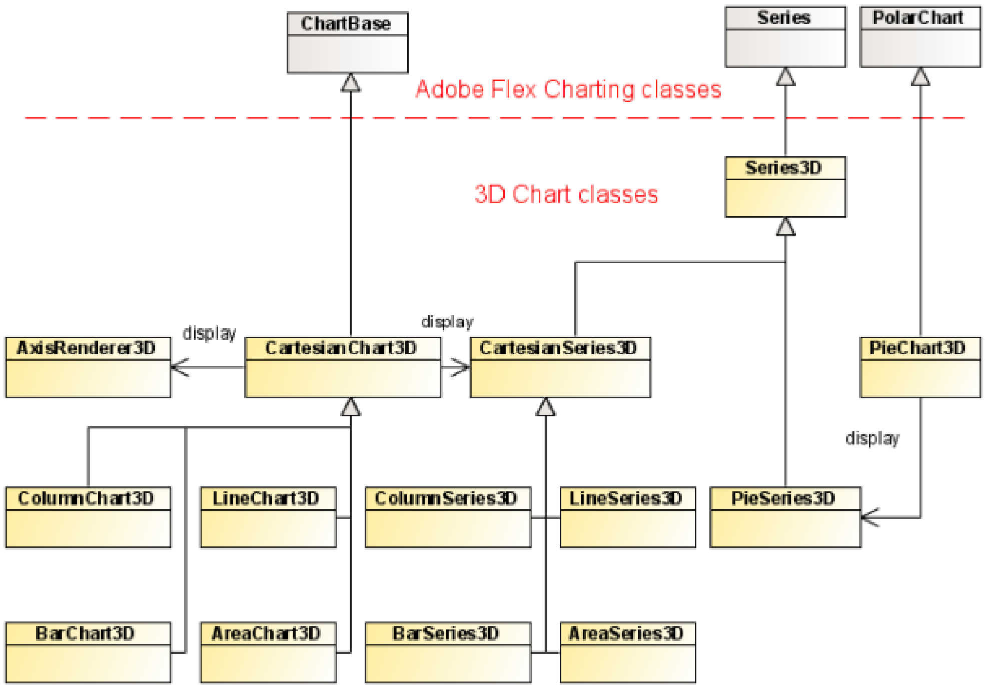
- Animation
- Legend
- Alternate axes (`LogAxis`, `DateTimeAxis`, `CategoryAxis`)

The following figure shows an example of an `AreaChart3D` control and a `PieChart3D` control.



3D chart architecture

The 3D chart classes rely on the Adobe® Flex® Builder™ Professional base classes as shown in the following figure. This allows you to reuse the same concepts, and in particular the Adobe Flex Builder Professional axes, legends, and effects, which can be used as is.



The 3D chart classes shown in the UML class diagram are described in the following table.

Main 3D chart classes

3D chart class	Description
`PieChart3D`	The class used to display 3D pie charts.
`CartesianChart3D`	The base class for all 3D Cartesian charts. The actual class to use depends on the kind of chart you want to display: ♦ `ColumnChart3D` ♦ `BarChart3D` ♦ `LineChart3D`

3D chart class	Description
	♦ `AreaChart3D`
`PieSeries3D`	The 3D pie series class that holds the data and styling configuration for the series.
`CartesianSeries3D`	The base class for all 3D Cartesian series. It holds the data and styling configuration for the series. The actual class to be used depends on the kind of series you want to display: ♦ `ColumnSeries3D` ♦ `BarSeries3D` ♦ `LineSeries3D` ♦ `AreaSeries3D`
`AxisRenderer3D`	The class in charge of rendering the 3D axes in Cartesian charts. It also holds the axis styling.

3D chart controls

Describes the 3D chart controls available.

In this section

Overview of IBM ILOG Elixir 3D chart controls
Explains the 3D chart controls available.

3D column chart
Describes a 3D column chart and gives examples.

3D bar chart
Describes a 3D bar chart and gives an example.

3D line chart
Describes a 3D line chart and gives an example.

3D area chart
Describes a 3D area chart and gives an example.

3D pie chart
Describes a 3D pie chart and gives an example.

3D composite chart
Describes a 3D composite chart and gives an example.

Overview of IBM ILOG Elixir 3D chart controls

In IBM ILOG Elixir a 3D chart control follows the architecture and paradigms of the Adobe® Flex® Builder™ Professional controls as closely as possible. You can use a 3D chart similarly to the Adobe Flex Builder Professional 2D equivalent type. For example, the `ColumnChart3D` class is very similar to the Adobe Flex `ColumnChart` class and the `PieChart3D` class is very similar to the Adobe Flex `PieChart` class. The types of 3D chart that you can create are as follows: column chart, bar chart, line chart, area chart, pie chart, composite charts.

The 3D chart examples given for each control all show the average temperatures for London, Sydney, and Beijing for the twelve months of the year.

3D column chart

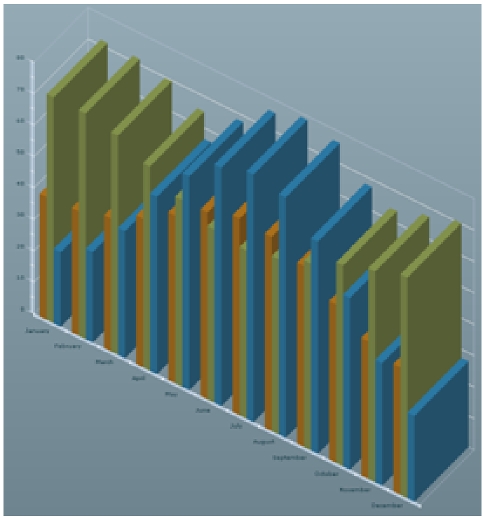
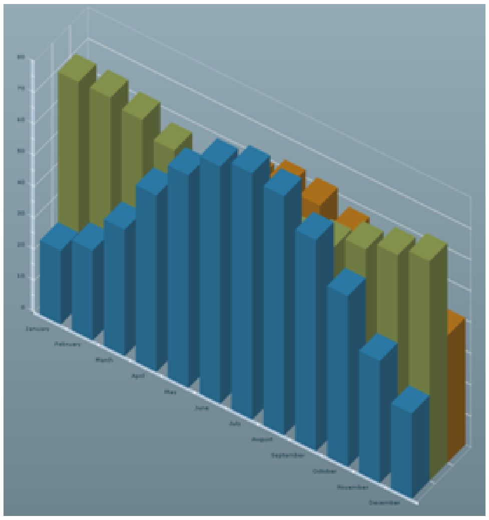
A 3D column chart contains a set of `ColumnSeries3D` classes that connects to the `dataProvider` of the chart. The following code shows the creation of a 3D column chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:ColumnChart3D width="100%" height="100%"
    dataProvider="{temperature}" showDataTips="true">
    <ilog:horizontalAxis>
      <mx:CategoryAxis categoryField="Month"/>
    </ilog:horizontalAxis>
    <ilog:series>
      <ilog:ColumnSeries3D yField="London" displayName="London" />
      <ilog:ColumnSeries3D yField="Sydney" displayName="Sydney" />
      <ilog:ColumnSeries3D yField="Beijing" displayName="Beijing" />
    </ilog:series>
  </ilog:ColumnChart3D>
</mx:Application>
```

The `ColumnChart3D` class has a `type` property that can be used to choose between several types of column arrangement. The following figures show the column chart coded above rendered with different `type` values. This shows how the `type` property affects the rendering.

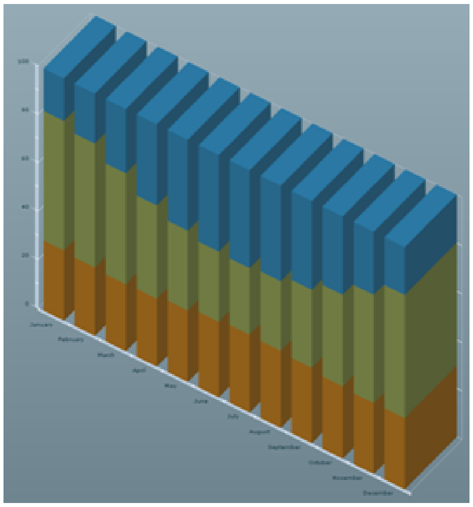
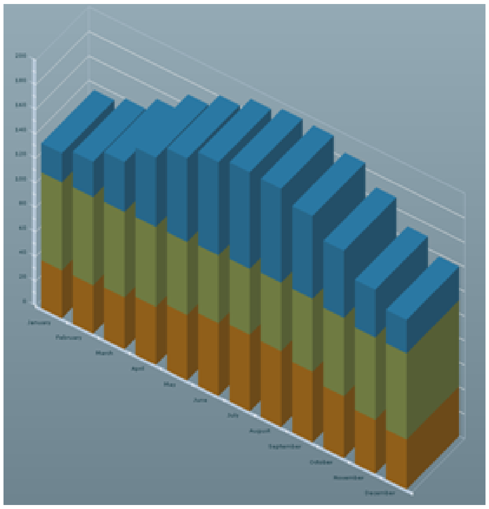
Example 1

Type values: type="clustered" (default value) and type="overlaid"

Example 2

Type values: type="100%" and type="stacked"

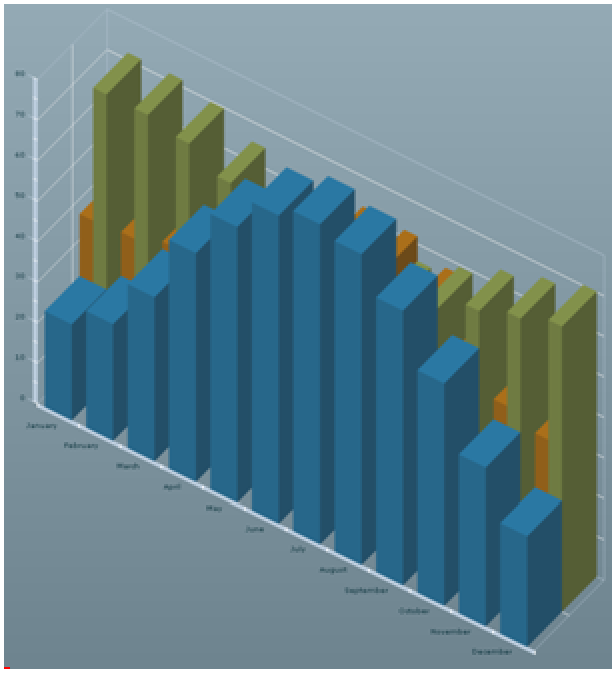
Complex charts

You can mix the four rendering styles to create more complex charts by using a `ColumnSet3D` object that groups columns in a similar way to how the Adobe® Flex® 2D `ColumnSet` object works on Adobe Flex 2D charts.

For example, the following code shows how to mix the `clustered` and `overlaid` types.

```
<ilog:ColumnChart3D width="100%" height="100%"
    dataProvider="{temperature}" showDataTips="true" type="overlaid">
   <ilog:horizontalAxis>
     <mx:CategoryAxis categoryField="Month"/>
   </ilog:horizontalAxis>
   <ilog:series>
     <ilog:ColumnSet3D type="clustered">
       <ilog:ColumnSeries3D yField="London" displayName="London"/>
       <ilog:ColumnSeries3D yField="Sydney" displayName="Sydney"/>
     </ilog:ColumnSet3D>
     <ilog:ColumnSeries3D yField="Beijing" displayName="Beijing"/>
   </ilog:series>
 </ilog:ColumnChart3D>
```

The following figure shows how a 3D chart that mixes `clustered` and `overlaid` types is rendered.



The `minField` property of a `ColumnSeries3D` class allows you to specify the field that is used to compute the minimum vertical value for the column. Instead of being drawn from the horizontal axis to the `y` value, it is drawn from the `minField` value to the `y` value.

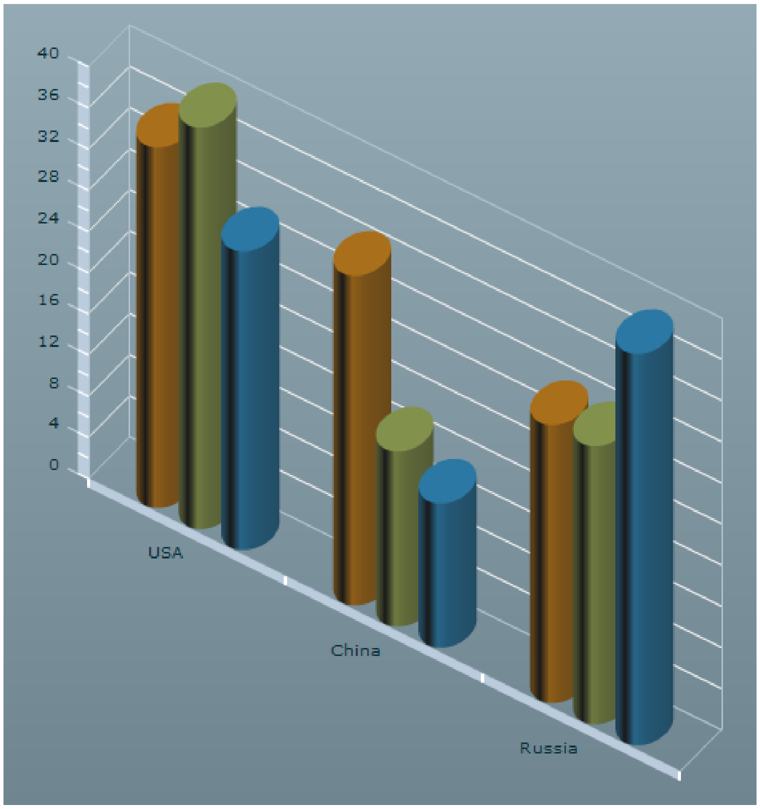
Cylinder representation

To switch from the default cuboid representation of the column to a cylinder representation, use the `form` attribute of the `ColumnSeries3D` class.

The following example in MXML shows the use of the `form` attribute in a column chart.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;
    [Bindable]
    private var medalsAC:ArrayCollection = new ArrayCollection( [
      { Country: "USA", Gold: 35, Silver:39, Bronze: 29 },
      { Country: "China", Gold: 32, Silver:17, Bronze: 14 },
      { Country: "Russia", Gold: 27, Silver:27, Bronze: 38 } ]);
    ]]>
  </mx:Script>
  <ilog:ColumnChart3D width="100%" height="100%" depth="10"
    showDataTips="true" dataProvider="{medalsAC}">
    <ilog:horizontalAxis>
      <mx:CategoryAxis categoryField="Country"/>
    </ilog:horizontalAxis>
    <ilog:series>
      <ilog:ColumnSeries3D xField="Country" yField="Gold" displayName="Gold"

        form="cylinder"/>
     <ilog:ColumnSeries3D xField="Country" yField="Silver" displayName="Silver"

        form="cylinder"/>
     <ilog:ColumnSeries3D xField="Country" yField="Bronze" displayName="Bronze"

        form="cylinder"/>
    </ilog:series>
  </ilog:ColumnChart3D>
</mx:Application>
```

This MXML example of cylinder representation for a column chart renders as shown in the following figure.

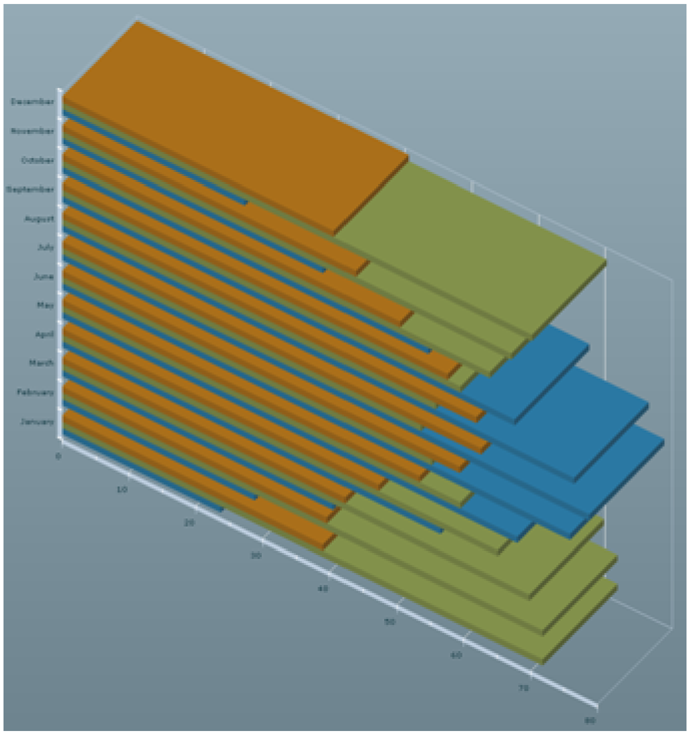

40
36
32
28
24
20
16
12
8
4
0
USA
China
Russia

3D bar chart

A 3D bar chart is very similar to a column chart except that it draws the bars horizontally instead of vertically. As such, the horizontal and vertical axes or values are inverted. A bar chart contains a set of `BarSeries3D` classes that connect to the chart `dataProvider`. The following code shows the creation of a 3D bar chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:BarChart3D width="100%" height="100%"
    dataProvider="{temperature}" showDataTips="true">
    <ilog:verticalAxis>
      <mx:CategoryAxis categoryField="Month"/>
    </ilog:verticalAxis>
    <ilog:series>
      <ilog:BarSeries3D xField="London" yField="Month" />
      <ilog:BarSeries3D xField="Sydney" yField=" Month " />
      <ilog:BarSeries3D xField="Beijing" yField="Month" />
    </ilog:series>
  </ilog:BarChart3D>
</mx:Application>
```

The following figure shows how the bar chart coded above is rendered.



The `BarChart3D` class shares its properties with `ColumnChart3D` and as such can use the `clustered` (default), `overlaid`, `stacked`, or `100%` types, or can be configured using `BarSet3D` objects.

In the same way as `ColumnSeries3D`, the `BarSeries3D` class supports a `minField` property to compute the base value of the bar. For more information, see *3D column chart*.

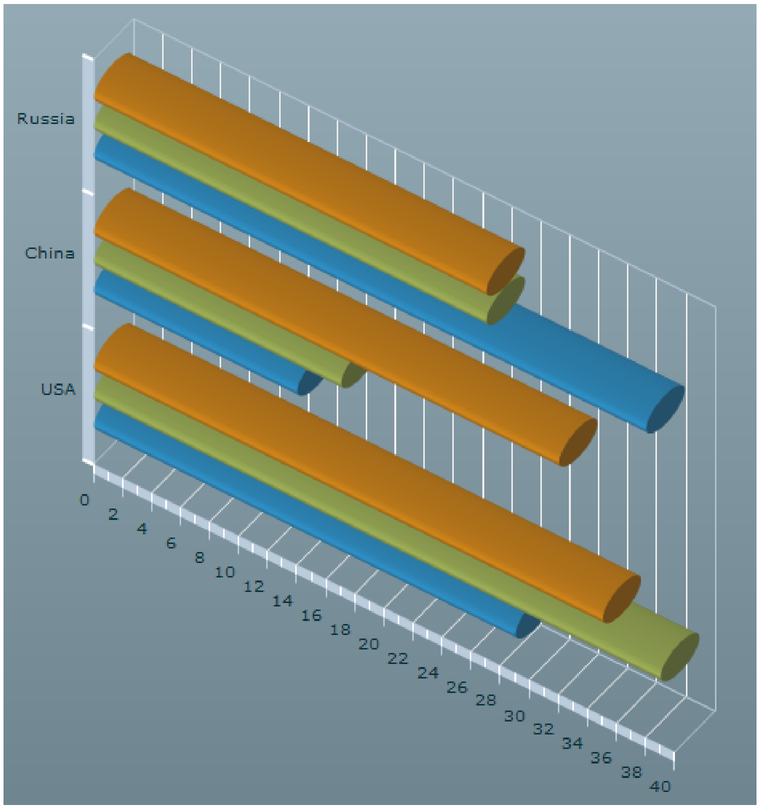
Cylinder representation

To switch from the default cuboid representation of the bar to a cylinder representation, use the `form` attribute of the `BarSeries3D` class.

The following example in MXML shows the use of the `form` attribute in a bar chart.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;
    [Bindable]
    private var medalsAC:ArrayCollection = new ArrayCollection( [
      { Country: "USA", Gold: 35, Silver:39, Bronze: 29 },
      { Country: "China", Gold: 32, Silver:17, Bronze: 14 },
      { Country: "Russia", Gold: 27, Silver:27, Bronze: 38 } ]);
    ]]>
  </mx:Script>
  <ilog:BarChart3D width="100%" height="100%" depth="10"
    showDataTips="true" dataProvider="{medalsAC}">
    <ilog:verticalAxis>
      <mx:CategoryAxis categoryField="Country"/>
    </ilog:verticalAxis>
    <ilog:series>
      <ilog:BarSeries3D yField="Country" xField="Gold" displayName="Gold"
        form="cylinder"/>
      <ilog:BarSeries3D yField="Country" xField="Silver" displayName="Silver"

        form="cylinder"/>
      <ilog:BarSeries3D yField="Country" xField="Bronze" displayName="Bronze"

        form="cylinder"/>
    </ilog:series>
  </ilog:BarChart3D>
</mx:Application>
```

This MXML example of cylinder representation for a bar chart renders as shown in the following figure.

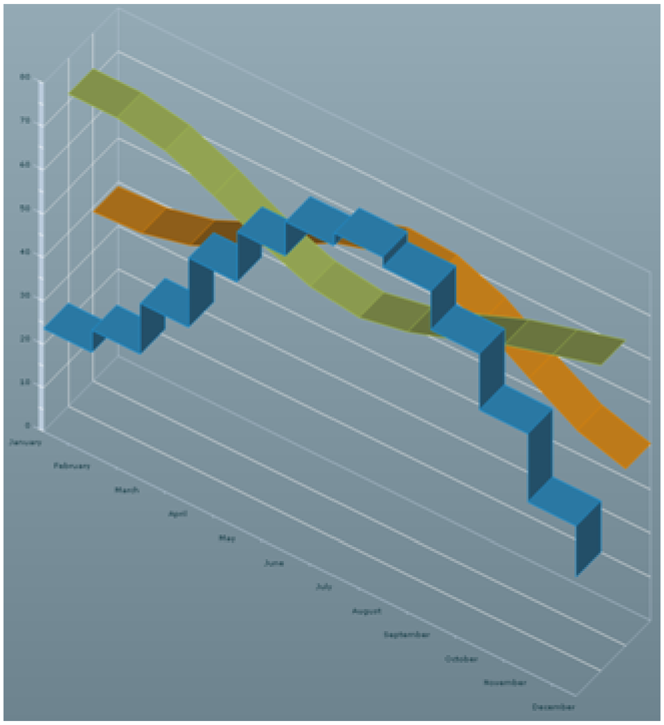


3D line chart

A 3D line chart contains a set of `LineSeries3D` classes that connect to the chart `dataProvider`. The following code shows the creation of a 3D line chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:LineChart3D width="100%" height="100%"
     dataProvider="{temperature}" showDataTips="true">
    <ilog:horizontalAxis>
      <mx:CategoryAxis categoryField="Month"/>
    </ilog:horizontalAxis>
    <ilog:series>
      <ilog:LineSeries3D yField="London" />
      <ilog:LineSeries3D yField="Sydney" />
      <ilog:LineSeries3D yField="Beijing" form="step"/>
    </ilog:series>
  </ilog:LineChart3D>
</mx:Application>
```

The following figure shows how the line chart coded above is rendered.



Note that the frontmost series is drawn by steps and not by segments. This feature is provided by the `form` style property of the `LineSeries3D` class.

The possible values are:

♦ `segment`: a direct segment is drawn from one point to another.

♦ `step`: two segments are drawn making a step from one point to another.

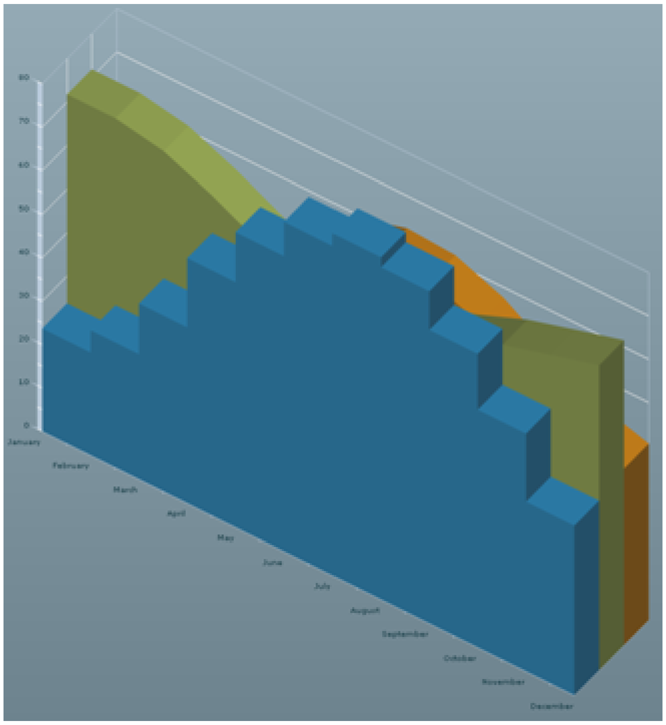
♦ `reverseStep`: the steps are formed in reverse.

3D area chart

A 3D area chart is very similar to a line chart except that it fills the chart area from the point to the base of the series. An area chart contains a set of `AreaSeries3D` classes that connect to the chart `dataProvider`. The following code shows the creation of a 3D area chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:AreaChart3D width="100%" height="100%"
     dataProvider="{temperature}" showDataTips="true">
    <ilog:horizontalAxis>
      <mx:CategoryAxis categoryField="Month"/>
    </ilog:horizontalAxis>
    <ilog:series>
      <ilog:AreaSeries3D yField="London" />
      <ilog:AreaSeries3D yField="Sydney" />
      <ilog:AreaSeries3D yField="Beijing" form="step"/>
    </ilog:series>
  </ilog:AreaChart3D>
</mx:Application>
```

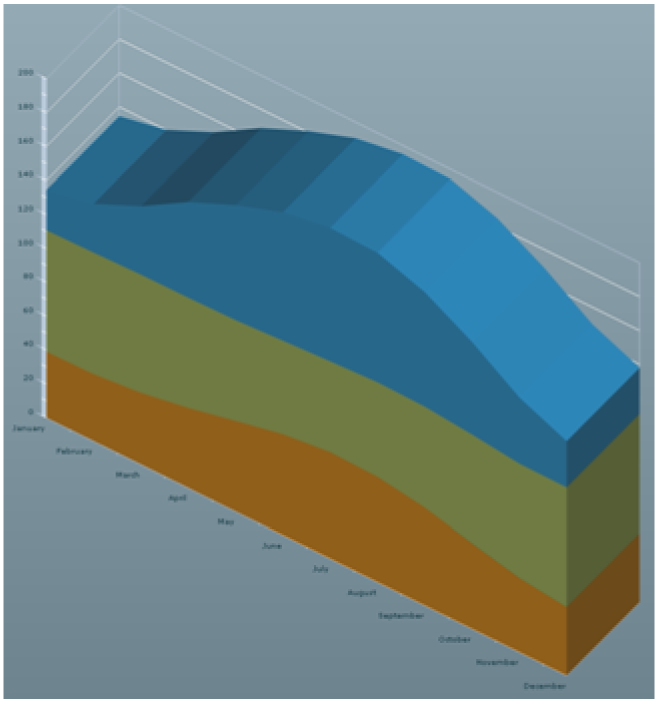
The following figure shows how the area chart coded above is rendered.



As for the `LineChart3D` class, the front most series is rendered with steps.

The `AreaChart3D` class supports an additional property compared to the `LineChart3D` class. This is the `type` property that can be `overlaid` (default), `stacked`, or `100%`.

The following figure shows the step form chart coded above rendered with a `stacked` type. This illustrates how the `type` property affects the rendering.

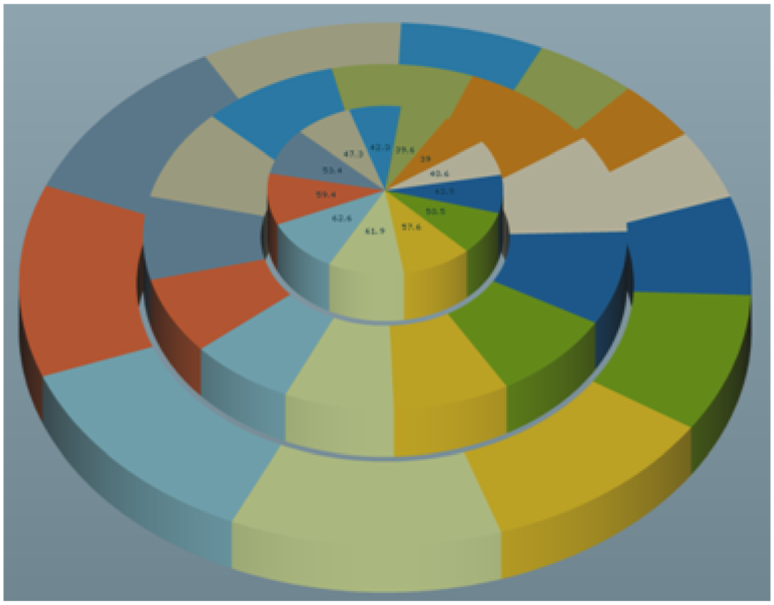

200
180
160
140
120
100
80
60
40
20
0
January
February
March
April
May
June
July
August
September
October
November
December

3D pie chart

A 3D pie chart contains a set of `PieSeries3D` classes that connect to the chart `dataProvider`. The following code shows the creation of a 3D pie chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:PieChart3D width="100%" height="100%"
     dataProvider="{temperature}" showDataTips="true">
    <ilog:series>
      <ilog:PieSeries3D nameField="Month" field="London"
         labelPosition="inside"/>
      <ilog:PieSeries3D nameField="Month" field="Sydney"/>
      <ilog:PieSeries3D nameField="Month" field="Beijing" />
    </ilog:series>
  </ilog:PieChart3D>
</mx:Application>
```

The following figure shows how the pie chart coded above is rendered.



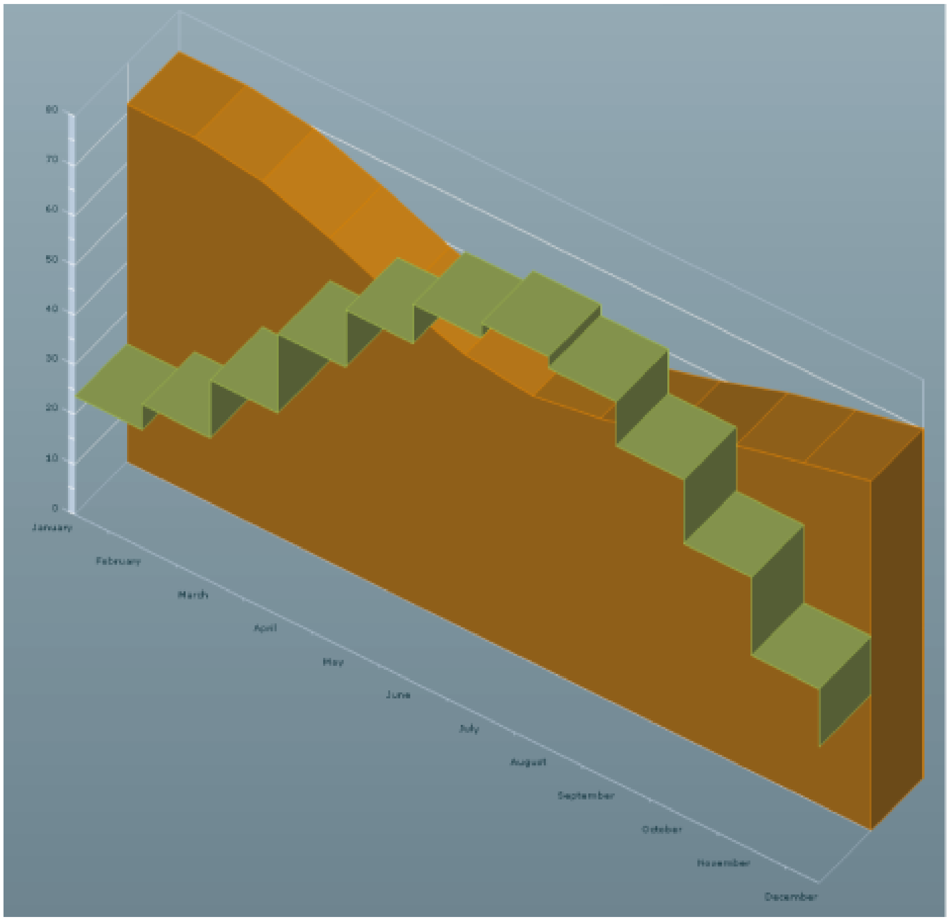
In this example, the innermost series uses the `labelPosition` style property to render labels inside the pie slices. Alternate values are `none` (default value) for no labels and `outside` to render labels outside the slices.

3D composite chart

You can create more complex 3D charts by including a given chart series of another type. For example, the following code displays a mixture of line and area series in a single 3D chart.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
  <![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
    public var temperature:ArrayCollection = new ArrayCollection([
      {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
      {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
      {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
      {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
      {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
      {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
      {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
      {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
      {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
      {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
      {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
      {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}]);
  ]]>
  </mx:Script>
  <ilog:LineChart3D width="100%" height="100%"
     dataProvider="{temperature}" showDataTips="true">
    <ilog:horizontalAxis>
      <mx:CategoryAxis categoryField="Month"/>
    </ilog:horizontalAxis>
    <ilog:series>
      <ilog:AreaSeries3D yField="Sydney" />
      <ilog:LineSeries3D yField="Beijing" form="step" />
    </ilog:series>
  </ilog:LineChart3D>
</mx:Application>
```

The following figure shows how the composite chart coded above is rendered.


80
70
60
50
40
30
20
10
0
January
February
March
April
May
June
July
August
September
October
November
December

Configuring 3D chart data

The 3D chart `dataProvider` accepts exactly the same data and can be configured in a similar manner to Adobe® Flex® charts.

For more information about 3D chart data,, see *Defining chart data* in *Getting Started with Flex 3*.

For more information on data provider types, see *Types of chart data* in the *Flex 3 Developer's Guide*.

To use the data from a data provider in your 3D chart control, you must map the field properties of the chart series to fields in the data provider.

For example, assuming your data provider has the following structure:

```
{x: 1000, y: 200}
```

You can map the `x` and `y` fields as shown in the following example:

```
<ilog:ColumnSeries3D xField="x" yField="y"/>
```

Styling 3D charts

Describes the common properties that can be used for styling the charts, the series, and the axes in 3D charts. These are in addition to the specific properties for each 3D chart control.

In this section

Chart styling
Describes the chart styling properties.

Series styling
Describes the properties for styling series.

Axis renderer styling
Describes the axis renderer styling properties.

Chart styling

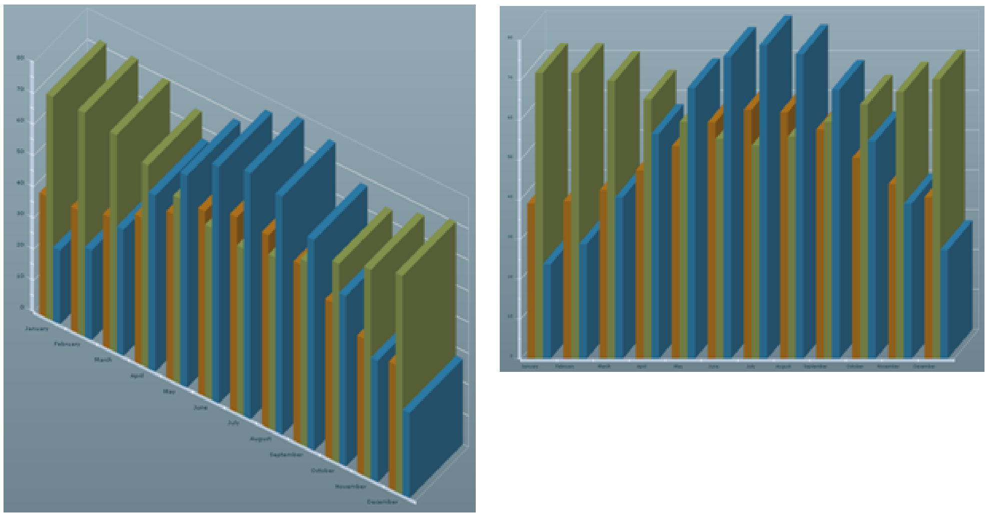
All 3D charts support properties that allow you to configure the way the 3D view is displayed.

The following properties apply to all 3D charts:

- ♦ The `rotationAngle` and `elevationAngle` properties allow you to decide the position of the eye looking at the 3D chart.
- ♦ The lighting properties `ambientLight`, `lightLatitude`, and `lightLongitude` allow you to specify how 3D objects are lit. The IBM ILOG Elixir lighting model is composed of:
  - A direct light that casts parallel rays. This light is located by spherical coordinates in the projected space. The default `lightLatitude` and `lightLongitude` values are equal to zero, which means that the light is located at the eye.
  - An ambient light that illuminates the 3D objects independently of the rendered faces. Changing the intensity prevents some parts of the chart from being too dark.

The following properties apply only to Cartesian charts:

- ♦ The `projectionType` property allows you choose how 2D values are projected in the 3D space. It can be either orthographic or oblique projection. The oblique projection preserves the orthogonality of the horizontal and vertical axes. The following figure shows the same chart rendered in orthographic (first chart) and oblique (second chart) projections:



- ♦ Some properties, related to grids and depth labels, allow you to specify how the walls around the charts are drawn.

For more information on 3D chart style properties, see `CartesianChart3D` and `PieChart3D`.

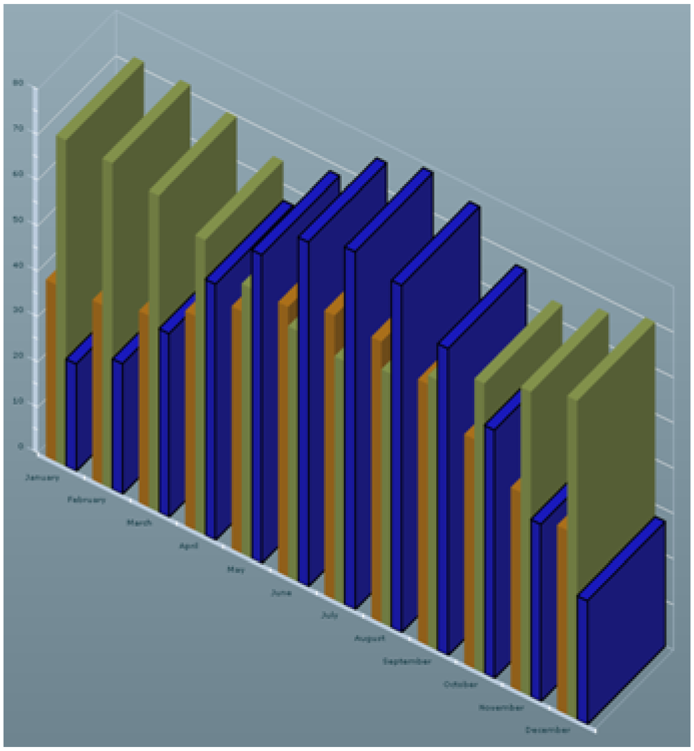
Series styling

In addition to the properties of the chart itself, some common properties are available for 3D chart series.

All Cartesian series support the `fill` and `stroke` style properties. You can use these properties to decide how you want to render a given series. For example, you can change the series definition of the `ColumnChart3D` example (see *3D column chart*) to the following:

```
<ilog:ColumnSeries3D yField="London" />
<ilog:ColumnSeries3D yField="Sydney" />
<ilog:ColumnSeries3D yField="Beijing">
  <ilog:fill>
    <mx:SolidColor color="blue"/>
   </ilog:fill>
   <ilog:stroke>
    <mx:Stroke color="black" weight="2"/>
   </ilog:stroke>
</ilog:ColumnSeries3D>
```

This causes the last series (Beijing) to be rendered in blue with a thick black border as shown in the following figure.



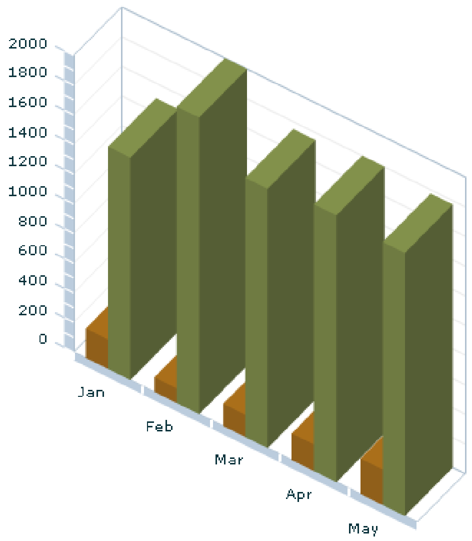
If you need each column or bar in a series, or each slice of a pie to be of a given color, you can either use the `fill` style property or give a `fillFunction` to the series that will be in charge of color computation.

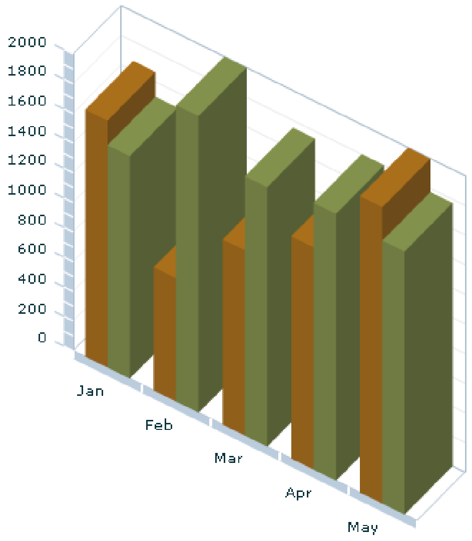
Another set of properties common to all the `Series3D`s are the various axis properties. For example, all `CartesianSeries3D` series have a `verticalAxis` and a `horizontalAxis` property. When these properties are specified, the series uses these axes as references instead of the chart axes. This allows you, for example, to compare data with different data ranges on a single chart.

The following example shows how to use this feature.

```
<?xml version="1.0"?>
<!-- Multiple Axis example to demonstrate the CartesianChart3D class. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

    <mx:Script>
        <![CDATA[

        import mx.collections.ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Profit: 200, Revenues: 1500 },
            { Month: "Feb", Profit: 100, Revenues: 2000 },
            { Month: "Mar", Profit: 150, Revenues: 1750 },
            { Month: "Apr", Profit: 180, Revenues: 1800 },
            { Month: "May", Profit: 240, Revenues: 1775} ]);
        ]]>
    </mx:Script>

    <mx:Panel title="Multiple Axis Example" height="100%" width="100%">

        <ilog:ColumnChart3D id="columnchart" height="100%" width="100%"
            paddingLeft="5" paddingRight="5"
            showDataTips="true" dataProvider="{expensesAC}">
            <ilog:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </ilog:horizontalAxis>

            <!-- Use two different axes to make sure the vertical values
                 of the two different series are spread on the whole
                 axis length. -->
            <ilog:series>
                <ilog:ColumnSeries3D yField="Profit" displayName="Profit">
                  <ilog:verticalAxis>
                    <mx:LinearAxis/>
                  </ilog:verticalAxis>
                </ilog:ColumnSeries3D>
               <ilog:ColumnSeries3D yField="Revenues" displayName="Expenses">

                  <ilog:verticalAxis>
                    <mx:LinearAxis/>
                  </ilog:verticalAxis>
                </ilog:ColumnSeries3D>
            </ilog:series>
        </ilog:ColumnChart3D>

    </mx:Panel>
</mx:Application>
```

The following figure shows the chart rendered without the axes specified on the series and then with the axes specified.





Axis renderer styling

In order to render axes, all `CartesianChart3D` classes use the `AxisRenderer3D` class. This class provides several properties that determine how the axis is rendered. To specify these properties you can use either the `horizontalAxisStyleName` or the `verticalAxisStyleName` style properties on the chart object, or use the `<ilog:AxisRenderer3D>` tag in your MXML as shown in the following example.

```
<?xml version="1.0"?>
<!-- Simple example to demonstrate the AxisRender3D class. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

    <mx:Script>
        <![CDATA[

        import mx.collections.ArrayCollection;

        [Bindable]
        private var stockDataAC:ArrayCollection = new ArrayCollection( [
            { Date: "4-Aug",  High: 37.98, Low: 36.56 },

            { Date: "5-Aug",  High: 37, Low: 36.48 } ]);
        ]]>
    </mx:Script>

    <mx:Panel title="AxisRenderer3D Example" height="100%" width="100%">

        <ilog:ColumnChart3D height="100%" width="100%"
            paddingRight="5" paddingLeft="5"
            showDataTips="true" dataProvider="{stockDataAC}">

            <ilog:verticalAxis>
                <mx:LinearAxis baseAtZero="false" />
            </ilog:verticalAxis>

            <ilog:horizontalAxis>
                <mx:CategoryAxis categoryField="Date" title="Date"/>
            </ilog:horizontalAxis>

            <ilog:horizontalAxisRenderer>
                <ilog:AxisRenderer3D canDropLabels="true"
                                     labelGap="15">
                  <ilog:fill>
                    <mx:SolidColor color="green"/>
                  </ilog:fill>
                </ilog:AxisRenderer3D>
            </ilog:horizontalAxisRenderer>

            <ilog:verticalAxisRenderer>
              <ilog:AxisRenderer3D>
                <ilog:fill>
                  <mx:SolidColor color="green"/>
                </ilog:fill>
              </ilog:AxisRenderer3D>
            </ilog:verticalAxisRenderer>
            <ilog:series>
                <ilog:ColumnSeries3D yField="High" minField="Low"/>

            </ilog:series>
        </ilog:ColumnChart3D>

    </mx:Panel>
</mx:Application>
```

Managing 3D chart events

The 3D chart controls manage user interaction events. The 3D charts component exposes all the Adobe® Flex® Builder™ Professional events, but does not extend them. These events are described in the section *About charting events* in the *Flex 3 Developer's Guide*.

The following code shows how to listen for an `itemClick` event on chart columns that call an `itemClickAction` function.

```
<ilog:ColumnChart3D height="100%" width="100%"
  itemClick="itemClickAction()"
  dataProvider="{stockDataAC}">
```

User interaction with 3D charts

3D charts support the Adobe® Flex® Builder™ Professional 3.0 interactions. These provide the chart interaction framework that allows you to select 3D chart items.

Selection

To allow selection on a given 3D chart, set the `selectionMode` property to `single` or to `multiple` if multiple items can be selected. The following code shows how to do this.

```
<ilog:ColumnChart3D height="100%" width="100%"
  selectionMode="multiple"
  dataProvider="{stockDataAC}">
```

You can select as shown in the following table.

Selection actions

Function	Action
Select	Click an item in the 3D chart.
Extend a selection	Hold down the CTRL key and click the items.
Reduce a selection	Hold down the CTRL key and click an already selected item to remove it from the selection.
Clear a selection	Click anywhere on the chart background, but not on 3D chart items.

Navigation

Keyboard accessibility is also provided for interactions with chart items. The keyboard actions for 3D charts are shown in the following table.

Keyboard actions

Keyboard	Action
Right/left arrow keys	Navigate through the items in a series.
Up/down arrow keys	Navigate through the series.

For more information, see *Handling user interactions with charts* and *Selecting chart items* in the *Flex 3 Developer's Guide*.

Using effects in a 3D chart

The 3D chart controls support the standard Adobe® Flex® Builder™ Professional series effects *slide*, *zoom*, and *interpolate*.

The SeriesSlide effect

The `SeriesSlide` effect allows you to make data slide in and out of a screen when the data changes. This creates animation effects using the two sets of values. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesSlide effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesSlide` effect.

```
<mx:SeriesSlide duration="1000" direction="down"
   elementOffset="0" id="slideDown" />
<ilog:ColumnSeries3D … showDataEffect="slideDown" hideDataEffect="slideDown"
/>
```

The SeriesZoom effect

The `SeriesZoom` effect allows you to implode and explode chart data in and out of the focal point that you specify to create chart animations. The effect is not set on the chart itself but on the series. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesZoom effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesZoom` effect.

```
<mx:SeriesZoom duration="1000" elementOffset="50"
    id="zoom" relativeTo="chart" />
<ilog:AreaSeries3D … showDataEffect="zoom" hideDataEffect="zoom" />
```

The SeriesInterpolate effect

The `SeriesInterpolate` effect allows you to move the graphics that represent the existing data in a series to new points. Instead of clearing the chart and then repopulating it, as with `SeriesZoom` and `SeriesSlide`, this effect keeps the data on the screen at all times.

You do not lose the context on data changes. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesInterpolate effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesInterpolate` effect.

```
<mx:SeriesInterpolate duration="1000" id="interpolate"
     elementOffset="10" />
<ilog:LineSeries3D … showDataEffect="interpolate" />
```

Calendar

Describes the use of IBM ILOG Elixir calendars with Adobe® Flex® 3.

In this section

Introduction to calendars
Describes the uses of calendars with an example

Calendar architecture
Describes the architecture of the calendar classes.

Creating a calendar control
Describes a calendar control and gives an example of the code.

Configuring calendar data
Describes the facilities for configuring your calendar data.

Configuring a calendar control
Explains how to configure the visible time range and the working and nonworking periods.

Calendar item renderers
Describes the use of the vertical item renderer, the horizontal item renderer, and the label item renderer, for displaying calendar information; the choice of item renderer depends on the event duration and the current mode of the calendar.

Styling a calendar
Describes the properties and methods that you can use to style different components of calendars.

Managing calendar events
Describes the management of events on calendars.

User interaction with calendars
Describes the interactions available with a calendar.

Event editing
Describes the event editing process and how to use calendar events associated with the editing of events when the user moves or resizes events in the calendar control by direct manipulation.

Recurring events
Explains how recurrence works and describes the support for recurring events.

Working with Date objects
Explains how to work with Date objects.

Introduction to calendars

Calendars are used to display events (tasks, appointments, meetings, and so on) in a specified period of time.

The calendar component is made up of:

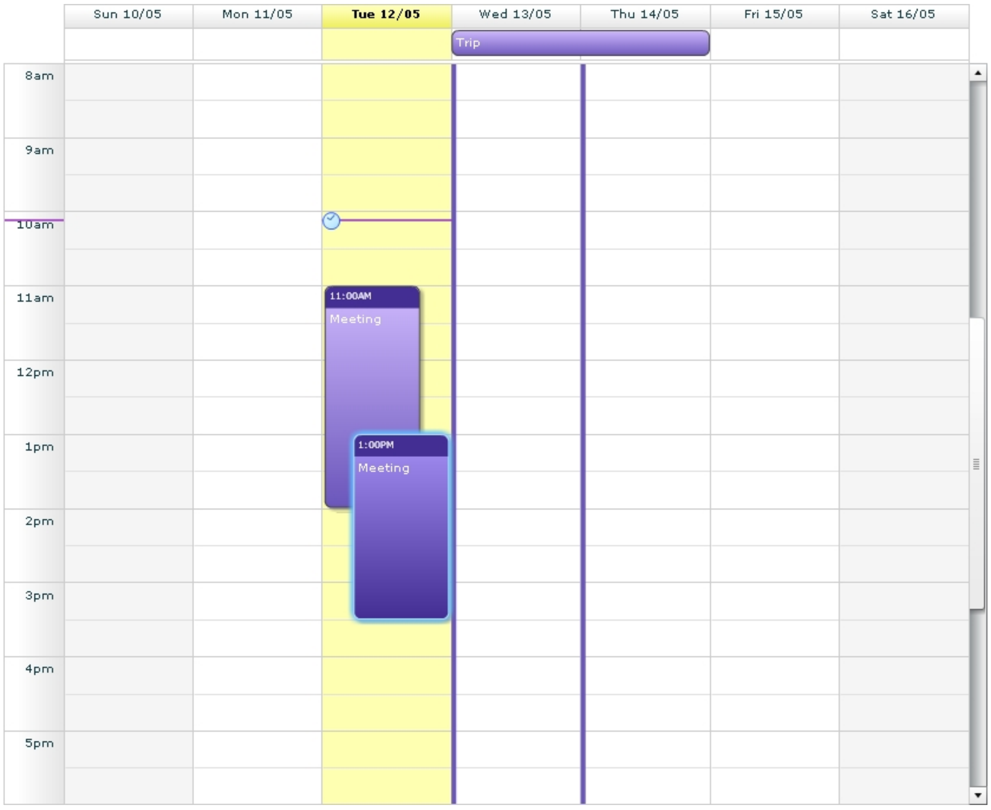
♦ A main sheet where the items are displayed.

♦ A secondary sheet on top of the main one which displays long events (at least 24 hours long) in day, work week, or week mode. In month mode, the secondary sheet is not displayed.

♦ A column header that displays the name of each day in month mode (such as `Tuesday`) or the label of each day in the other modes (such as `Tuesday, December 16`).

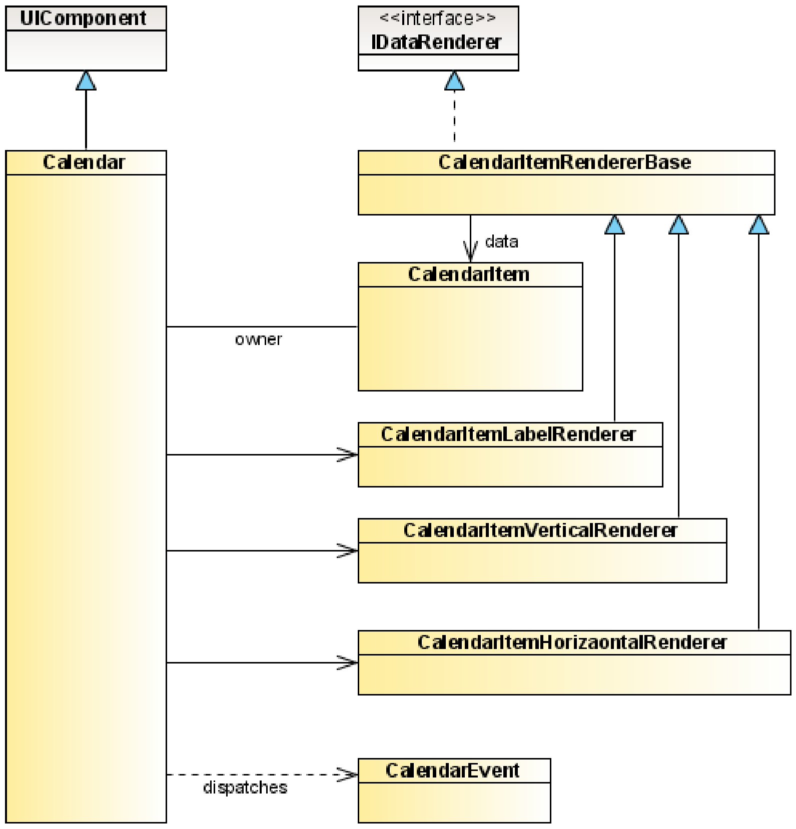
♦ A row header that shows week numbers or hours of the day depending on the current mode.

An example of a calendar in week mode is shown in the following figure.



Calendar architecture

The calendar architecture is shown in the following figure.



The Calendar classes shown in the UML diagram are described in the following table.

Class	Description
`Calendar`	The calendar component. It extends the UIComponent class. It is the top level object to be used in MXML.
`CalendarEvent`	Defines the types of event displayed by the calendar.
`CalendarItemRendererBase`	The base class of the default calendar item renderers.
`CalendarItemVerticalRenderer`	The default item renderer that displays short events in day, work week, or week modes.
`CalendarItemHorizontalRenderer`	The default item renderer that displays long events in any mode.
`CalendarItemLabelRenderer`	The default item renderer that displays short events in month mode.

Creating a calendar control

You can define a calendar control in MXML using the `<ilog:Calendar>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value. A calendar control displays a list of events using a `dataProvider` as shown in the following code.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <ilog:Calendar width="100%" height="100%">
    <ilog:dataProvider>
      <mx:XML>
       <event startTime="2008/4/16 10:00" endTime="2008/4/16 16:00" summary="Do
 something"/>
      </mx:XML>
    </ilog:dataProvider>
  </ilog:Calendar>
</mx:Application>
```

Configuring calendar data

Describes the facilities for configuring your calendar data.

In this section

Calendar data providers
Describes the possible data providers for calendars.

Specifying calendar data with field mappings
Describes the properties for mapping calendar event information and how to map them.

Specifying calendar data with custom functions
Describes how to provide calendar data with custom functions.

Multiple calendars
Explains how you can have several calendars within one control.

Calendar data providers

A Calendar component gets its data from a data provider, which contains a list of data items. Each data item represents an event.

A calendar data provider can be one of the following types:

XML
A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag or an XML or XMLList object.

IList or ICollectionView
An object that implements the `IList` or `ICollectionView` interface, such as an instance of the `ArrayCollection` class or the `XMLListCollection`class, with a data provider that conforms to the structure specified in either of these items.

Other objects
An array of items or an object.

When setting the `dataProvider` property the `Calendar` control modifies its internal representation of the task data as follows:

- ♦ XML or XMLList are wrapped in an instance of `XMLListCollection`.
- ♦ `IList` or `ICollectionView` are used directly.
- ♦ An `Array` object is wrapped in an instance of `ArrayCollection`.
- ♦ An ActionScript® object of any other data type is wrapped in an `ArrayCollection` using an `Array` containing the object as its sole entry.

For example, you pass an `Array` to the `dataProvider` property. If you read the data back from the `dataProvider` property it is returned as an instance of `ArrayCollection`.

The best practice when your event data can change dynamically is to use a collection, which provides the necessary notifications of changes.

Specifying calendar data with field mappings

In addition to specifying the `dataProvider` of the Calendar control, to display (render) the events you must map the fields of data items (start time, end time, and summary) to fields that exist in the `dataProvider` by setting field properties in the control. In the data provider, a field is either a property (if the data provider items are ActionScript® objects) or an attribute (if the data provider items are XML objects).

The following table shows the field properties in a Calendar control that contain the field names used to map data. In each case, the first default value of the field name is for an ActionScript object and the second (with @ prefix) is for an XML object.

Field properties for calendar events

Property	Default value		Purpose
calendarField	calendar or @calendar	Optional	The value of the field whose name is stored in this property is used as the identifier of the calendar that contains this event.
endTimeField	endTime or @endTime	Mandatory	The value of the field whose name is stored in this property is used as the end time of each data item.
startTimeField	startTime or @startTime	Mandatory	The value of the field whose name is stored in this property is used as the start time of each data item.
summaryField	summary or @summary	Mandatory	The value of the field whose name is stored in this property is used to display the summary of each data item.

The calendar also defines other fields that are not used by the default item renderers but are available to simplify the creation of advanced custom item renderers.

Field properties for advanced custom item renderers

Property	Default value	Purpose
categoriesField	categories or @categories	The value of the field whose name is stored in this property is used to display the categories of each data item.
descriptionField	description or @description	The value of the field whose name is stored in this property is used to display the description of each data item.
locationField	location or @location	The value of the field whose name is stored in this property is used to display the location of each data item.
statusField	status or @status	The value of the field whose name is stored in this property is used to display the status of each data item.

In the following sample the data has the start time and the end time in the start and end fields respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <ilog:Calendar width="100%" height="100%"
                 startTimeField="@start" endTimeField="@end"
                 date="{new Date(2008,0,14)}">
    <ilog:dataProvider>
      <mx:XMLList>
        <event summary="Event" start="2008/01/14 14:00:00" end="2008/01/14
17:00:00"/>
      </mx:XMLList>
    </ilog:dataProvider>
  </ilog:Calendar>

</mx:Application>
```

Specifying calendar data with custom functions

If the data values for a field in a calendar control are not directly available in one of the fields of the data provider items, you can specify a custom function to provide the `Calendar` control with the required information.

If a custom function and a field are both specified, the function takes precedence over the field.

The function properties used to specify custom functions are listed in the following table.

Functions and their corresponding field names

Function property	Corresponding field name
calendarFunction	calendar
categoriesFunction	categories
descriptionFunction	description
endTimeFunction	endTime
locationFunction	location
startTimeFunction	startTime
summaryFunction	summary
statusFunction	status

The following example shows how you can replace the default date parsing function by your own to read the start and end time in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <mx:Script>
    <![CDATA[

      public function parseDate(value:String):Date {
        return new Date(value);
      }

      public function startTimeFunction(item:Object):Date {
        return parseDate(item.@startTime);
      }

      public function endTimeFunction(item:Object):Date {
        return parseDate(item.@endTime);
      }

    ]]>
  </mx:Script>
  <ilog:Calendar width="100%" height="100%"
                 startTimeFunction="{startTimeFunction}"
                 endTimeFunction="{endTimeFunction}">
    <ilog:dataProvider>
      <mx:XML>
        <event startTime="4/16/2008/ 10:0:0" endTime="4/16/2008/ 16:0:0"
summary="Do something"/>
      </mx:XML>
    </ilog:dataProvider>
  </ilog:Calendar>
</mx:Application>
```

Multiple calendars

The Calendar control can display several calendars, for example, a personal calendar and a professional one as shown in the following code example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

<ilog:Calendar width="100%" height="100%"
date="{new Date(2008,9,11)}">
<ilog:dataProvider>
<mx:XMLList>
<event startTime="2008/10/8 10:00" endTime="2008/10/8 12:00"
summary="Professional Event" calendar="pro"/> <event startTime="2008/10/8
11:00" endTime="2008/10/8 13:00"
summary="Personal Event" calendar="per"/>

</mx:XMLList>
</ilog:dataProvider>
</ilog:Calendar>
</mx:Application>
```

The `dataProvider` is a one-dimensional (not hierarchical) list of events but each event can be associated with a potentially different calendar.

The `calendarField` or `calendarFunction` property is used to determine the unique identifier of the calendar that contains the event. If no calendar is defined, the event is associated with a default calendar.

The calendar identifier for an event is used by the default item renderers to represent each event with the color associated with its calendar.

Configuring a calendar control

Explains how to configure the visible time range and the working and nonworking periods.

In this section

Specifying the visible time range
Explains how to set the visible time range and the resulting mode.

Defining working and nonworking periods
Explains how to set properties to define working and nonworking periods shown in the grid.

Specifying the visible time range

By default, the calendar displays the current week.

You can specify the visible time range in two different ways:

Specify a date and a mode

Set the `date` property to specify a reference date.

The control sets the visible range for the view mode according to the following logic:

- ♦ day mode: Displays the specified date.
- ♦ week mode: Displays the week that contains the specified date. This is the default mode.
- ♦ work week mode: Displays the work week corresponding to the week that contains the specified date. The `workWeekRange` property defines the first and last day of the work week.
- ♦ month mode: Displays the month that contains the specified date.

Specify a start and end date

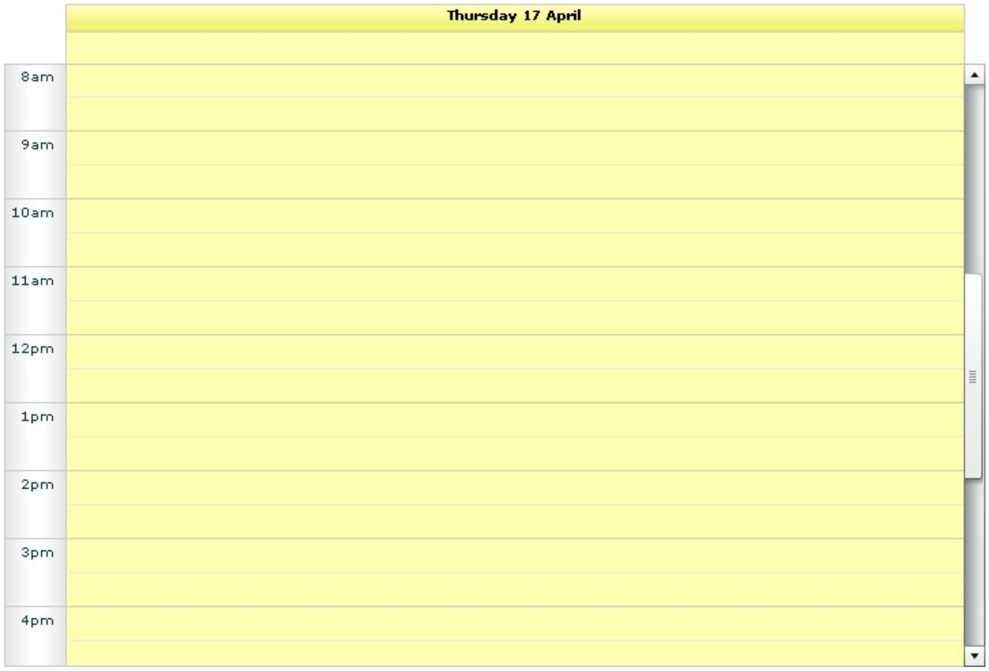
If the `date` property is not set (that is, is `null`), the control uses the `startDate` and `endDate` properties to set the visible time range. The time of day (hours, minutes, seconds, milliseconds) is not taken into account. The end date is included in the visible time range.

The view mode is then set according to the duration of the time range as follows:

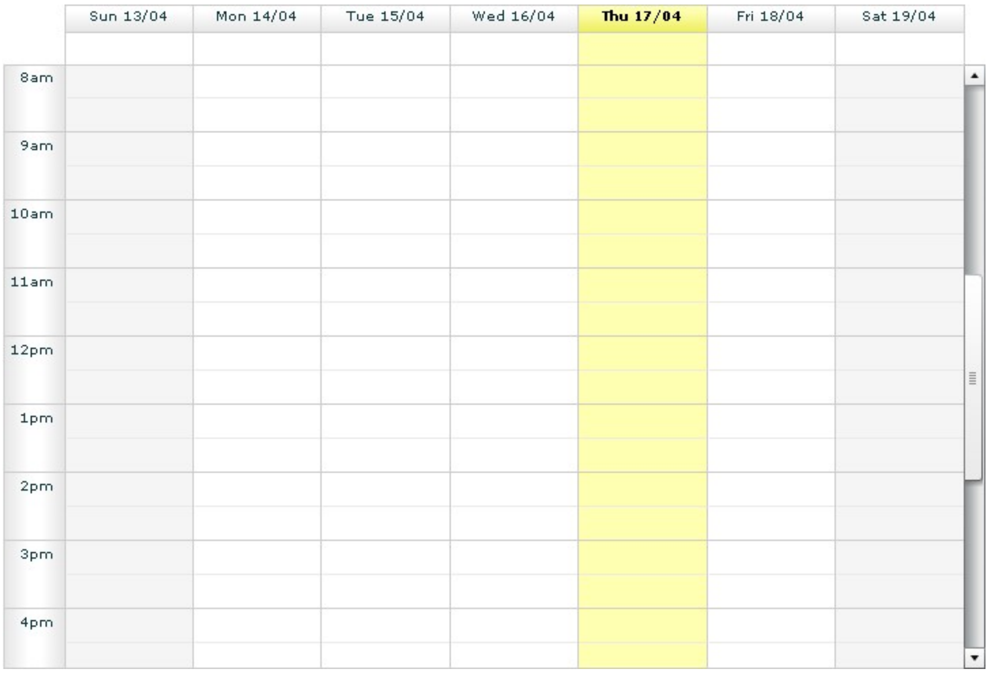
- ♦ 1 day: The view mode is set to day.
- ♦ 2 to 7 days: The view mode is set to week. In week mode, the specified days are displayed, not necessarily the whole week.
- ♦ More than 8 days: The view mode is set to month. The specified time range is rounded up to show whole weeks; dates that are out of the specified time interval are shown in a different color.

At most 6 weeks are shown in the Calendar control; if more days are requested, the period will be truncated.

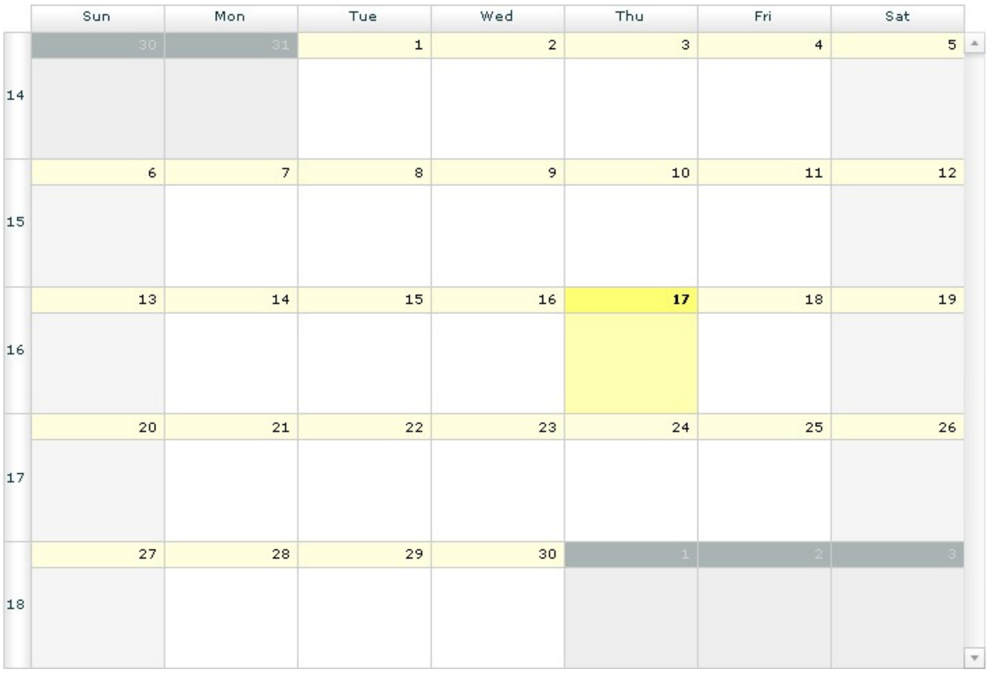
The following figures show the day, week, and month view modes respectively.



Calendar in day mode showing hours



Calendar in week mode, showing hours of each day



Calendar in month mode showing days (no hours)

By default, in work week, week, or day mode, the calendar shows the events between 8:00am and 6:00pm. You can change the displayed time of day using the `startDisplayedTime` and `endDisplayedTime` properties of the calendar.

The following sample shows how to display the events between 9:00am and 4:00pm at the start of the application.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <ilog:Calendar width="100%" height="100%"
                 startDisplayedTime="[9,0]" endDisplayedTime="[16,0]" />

</mx:Application>
```

The `scrollToTime` method allows you to make the application scroll to the specified time (with an optional animation).

The zoom method allows you to zoom in or out (increase/decrease the height of an hour).

Defining working and nonworking periods

The Calendar control displays a grid behind the events. This grid shows the working and nonworking periods.

The working and nonworking periods are defined by the following properties of the Calendar object.

nonWorkingDays
: The list of days of the week that are nonworking days.

workingTimes
: The ranges of working time for each day of the week. Each day of the week can have no, one, or several ranges of working time.

nonWorkingRanges
: Additional arbitrary ranges of nonworking time. These ranges are defined as full days or ranges of time of any duration. The nonworking time periods defined by this property take precedence over the working and nonworking times defined by the other properties.

The following example defines Saturday and Sunday as the nonworking days of the week.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:Calendar width="100%" height="100%" nonWorkingDays="{[0,6]}" />
</mx:Application>
```

The following example defines the working times as follows:

♦ For all working days but Friday:
  - from 8:00 to 11:45
  - from 12:45 to 17:30

♦ For Friday:
  - from 8:00 to 11:45
  - from 12:45 to 16:30

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:Calendar width="100%" height="100%">
    <ilog:workingTimes>
      <mx:Array>
        <mx:Object rangeStart="8:00" rangeEnd="11:45" />
        <mx:Object rangeStart="12:45" rangeEnd="17:30" />
        <mx:Object days="[5]" rangeStart="8:00" rangeEnd="11:45" />
        <mx:Object days="[5]" rangeStart="12:45" rangeEnd="16:30" />
      </mx:Array>
    </ilog:workingTimes>
  </ilog:Calendar>
</mx:Application>
```

The following example defines 2008-04-14 as a nonworking day, and the range from 2008-04-15 12:00:00 to 2008-04-18 14:30:00 as nonworking time.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:Calendar width="100%" height="100%" mode="month"
                 nonWorkingRanges="{[ new Date(2008,3,14),
                                      {rangeStart: new Date(2008,3,15,12,0),
                                       rangeEnd: new Date(2008,3,18,14,30}]"
/>
</mx:Application>
```

Calendar item renderers

Describes the use of the vertical item renderer, the horizontal item renderer, and the label item renderer, for displaying calendar information; the choice of item renderer depends on the event duration and the current mode of the calendar.

In this section

Vertical item renderer
Describes the use of the vertical item renderer and shows the default vertical item renderer.

Horizontal item renderer
Describes the use of the horizontal item renderer and shows the default horizontal item renderer.

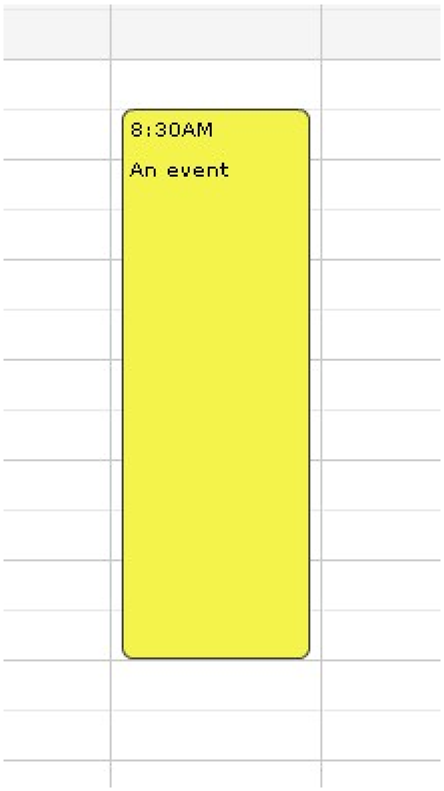
Label item renderer
Describes the use of the label item renderer and shows the default label item renderer.

Custom item renderers
Describes how to develop a custom item renderer.

Vertical item renderer

A vertical item renderer is used in day or week mode to display an event that lasts less than or equal to 24 hours. Such an event is known as a *short event*.

The following figure shows the default vertical item renderer.



The default item renderer displays the start hour. It also displays the summary of the event if there is enough space.

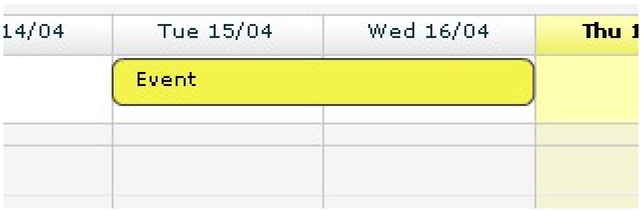
The following example shows how to set and configure a vertical item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:Calendar width="100%" height="100%">
    <ilog:itemVerticalRenderer>
      <mx:Component>
        <ilog:CalendarItemVerticalRenderer color="#FF0000" />
      </mx:Component>
    </ilog:itemVerticalRenderer>
  </ilog:Calendar>
</mx:Application>
```

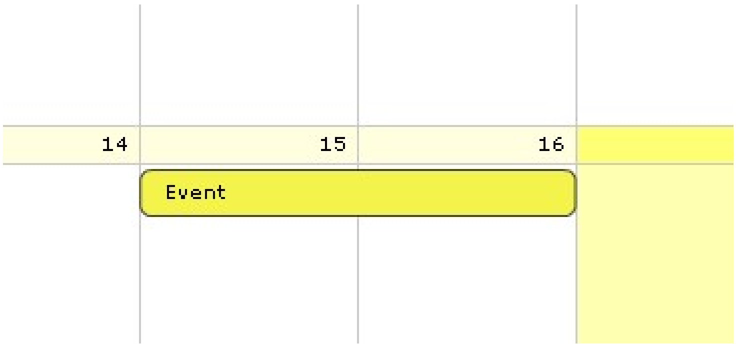
Horizontal item renderer

A horizontal item renderer is used in week or month mode to display an event which lasts for more than 24 hours. Such an event is known as a *long event*.

The following figures show the default horizontal item renderer in week mode and month mode respectively.



The default horizontal item renderer showing an event that spans two days in week mode



The default horizontal item renderer showing an event that spans two days in month mode

If the event is partially represented, the default item renderer displays:

♦ For an all-day event, the summary and arrows.

♦ For an event that is not all-day, the start and end time, the summary of the event, and arrows.

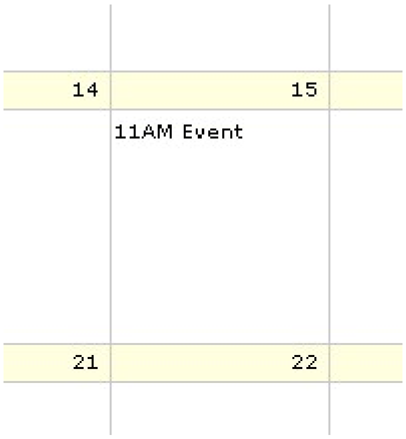
The following example shows how to set and configure a horizontal item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <ilog:Calendar width="100%" height="100%">
    <ilog:itemHorizontalRenderer>
      <mx:Component>
        <ilog:CalendarItemHorizontalRenderer color="#FF0000" />
      </mx:Component>
    </ilog:itemHorizontalRenderer>
  </ilog:Calendar>

</mx:Application>
```

Label item renderer

A label item renderer is used in month mode to display an event that lasts less than or equal to 24 hours. Such an event is known as a *short event*.

The following figure shows the default label item renderer.



The default label item renderer displays the start hour of the event. It also displays the summary of the event if there is enough space.

The following example shows how to set and configure a label item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <ilog:Calendar width="100%" height="100%">
    <ilog:itemLabelRenderer>
      <mx:Component>
        <ilog:CalendarItemLabelRenderer color="#FF0000" />
      </mx:Component>
    </ilog:itemLabelRenderer>
  </ilog:Calendar>

</mx:Application>
```

The label item renderer does not have a background and border by default. As a consequence, the color defined by the `itemColorFunction` will not be used by the label item renderer. To see how to set a background and border, refer to the `useBorder` property.

Custom item renderers

A `Calendar` control can use custom renderers to display the data items.

A custom item renderer must:

- ♦ Extend a `UIComponent` subclass
- ♦ Implement the `IListItemRenderer` interface

Before you decide whether and how to develop a custom item renderer, make sure you read about Custom Component Development in the Adobe® Flex® documentation.

The content of the `data` property for an item renderer is of the type `CalendarItem`.

The `CalendarItem` type gives access to:

- ♦ The data fields to display in the item renderer
- ♦ The calendar control

The calendar provides the `CalendarItemRendererBase` class, which is the base class of the supplied item renderers, to make the creation of efficient calendar item renderers easier in ActionScript® .

The `CalendarItemRendererBase` class:

- ♦ Declares the usual style properties
- ♦ Declares the usual subcomponents (but does not create them)
- ♦ Manages a border skin to render the border and the background of the item renderer
- ♦ Provides an API to determine the state of the represented item easily (selected, highlighted, being edited, being animated, and so on)

A subclass typically:

- ♦ Creates the required subcomponents in the `createChildren` method
- ♦ Implements the `measure` method for the measure constraint (see the table that follows)
- ♦ Uses the `data` property to set the value to display for each subcomponent, in the `commitProperties` method
- ♦ Applies the style properties on subcomponents
- ♦ Lays out the subcomponents in the `updateDisplayList` method

The default item renderer implementations give examples of this behavior.

For the `Calendar` control to lay out the item renderer, some item renderers must have a valid measure. The following table lists the measure constraint for each type of item renderer.

Type	Measure constraint
Vertical	No need to measure
Horizontal	Need of a valid measured height
Label	Need of a valid measured height

Styling a calendar

Describes the properties and methods that you can use to style different components of calendars.

In this section

Grids
Describes how to customize the colors of the grid.

Column and row headers
Describes how to customize the colors of column and row headers.

Items
Describes how to customize the colors of items.

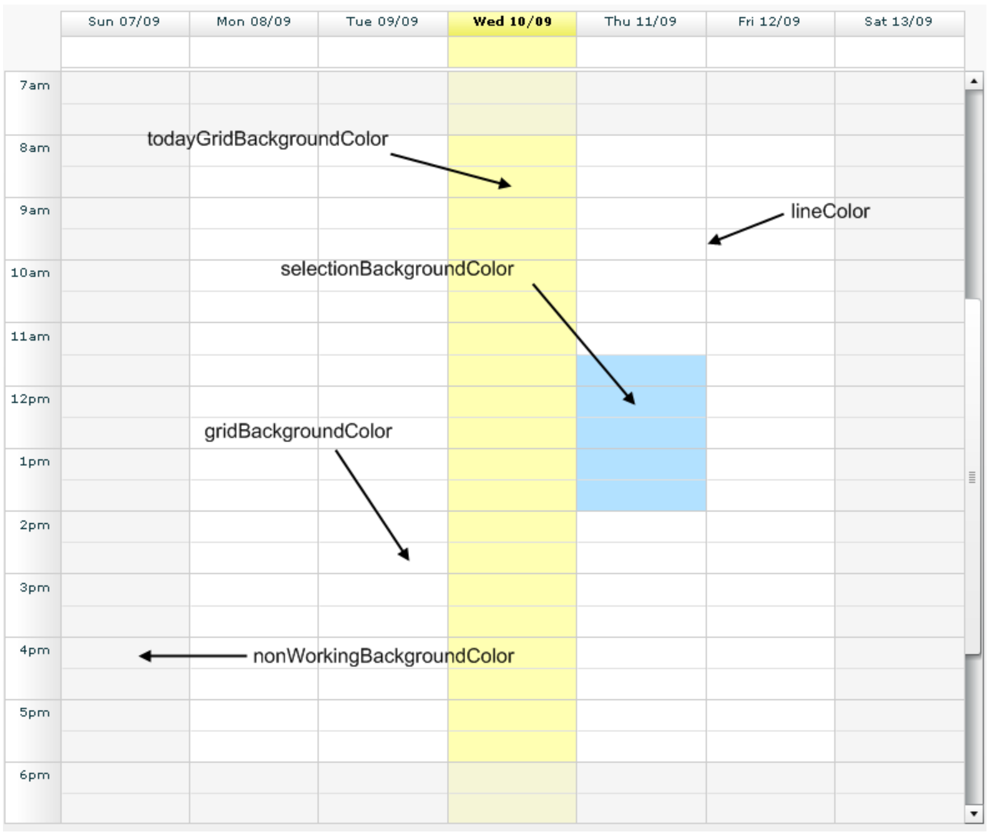
Current time indicator
Describes how to customize the current time indicator.

Grids

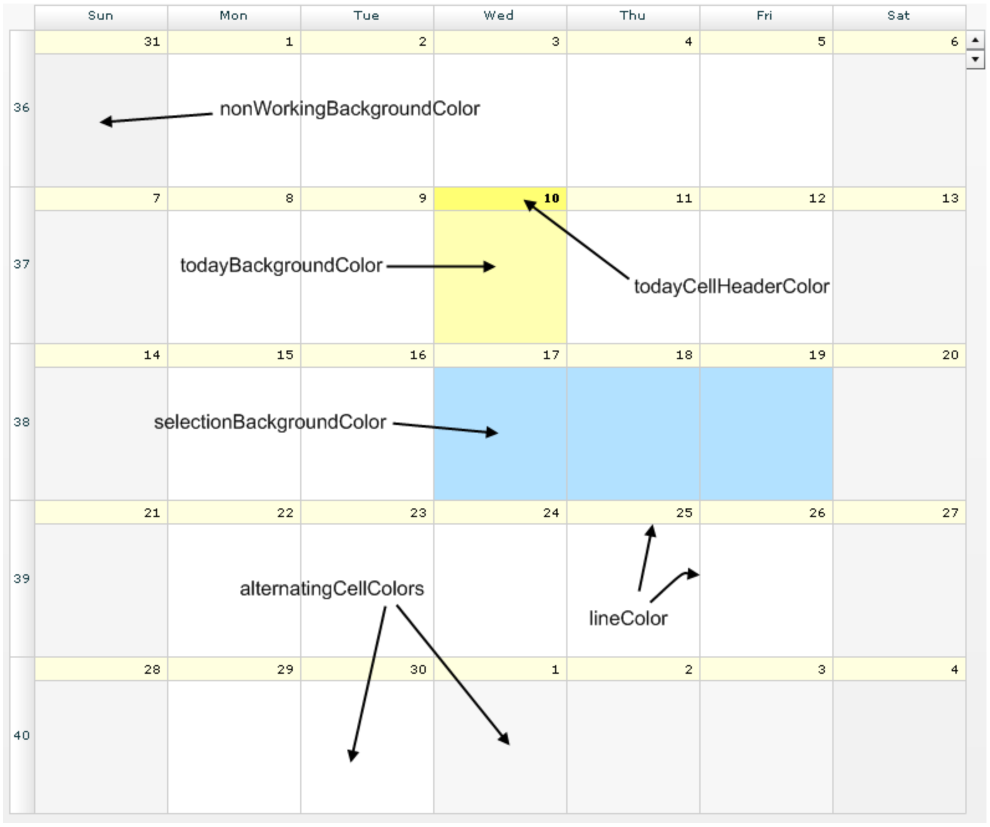
The calendar displays a grid filled with different colors. The following properties are used to customize the grid.

Color property	Description
alternatingCellColors	In month mode, two colors used to display the background of odd and even months.
cellHeaderColor	In month mode, the color of the cell header.
disabledBackgroundColor	In month mode, the color used to fill the cells of dates out of the requested time range.
gridBackgroundColor	In any mode, the grid background color.
lineColor	The color of the grid lines and of the borders of the headers.
nonWorkingBackgroundColor	The overlay color that represents nonworking periods of time.
selectionBackgroundColor	The overlay color that represents the time range selection.
todayCellHeaderColor	In month mode, the color of the cell header for the current day.
todayGridBackgroundColor	In any mode, the grid background color of the current day.

The following figure shows how the grid color properties are applied in week mode.



The following figure shows how the grid color properties are applied in month mode.


Sun
Mon
Tue
Wed
Thu
Fri
Sat
nonWorkingBackgroundColor
todayBackgroundColor
todayCellHeaderColor
selectionBackgroundColor
alternatingCellColors
lineColor

Column and row headers

Fill

The fill for column and row headers is delegated to a skin specified in the `headersSkin` style property.

By default the skin is an instance of `CalendarHeaderSkin`. This class uses the `headersBackgroundColors` and `todayHeadersBackgroundColors` style properties to fill the headers.

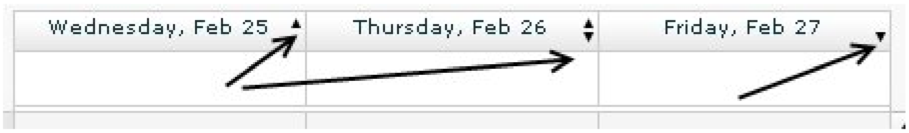
The `selectionColor` and `rollOverColor` properties allow you to customize the cell background color for the row and column headers when they are selected and highlighted respectively.

Column item indicators

In day, work week, or week mode, the column header can display an indicator when an item is present in the column but is not visible. There may be items earlier than the displayed time range in the day or later than this range, or earlier and later than this range.

The `showItemIndicators` style property specifies whether or not to display such indicators. You can customize the background color of the indicators through the `itemIndicatorsColor` style property.

The following figure shows the earlier, earlier and later, and later indicators.



Items

The default item renderers use the `getItemColor` method of the Calendar control to set their background color.

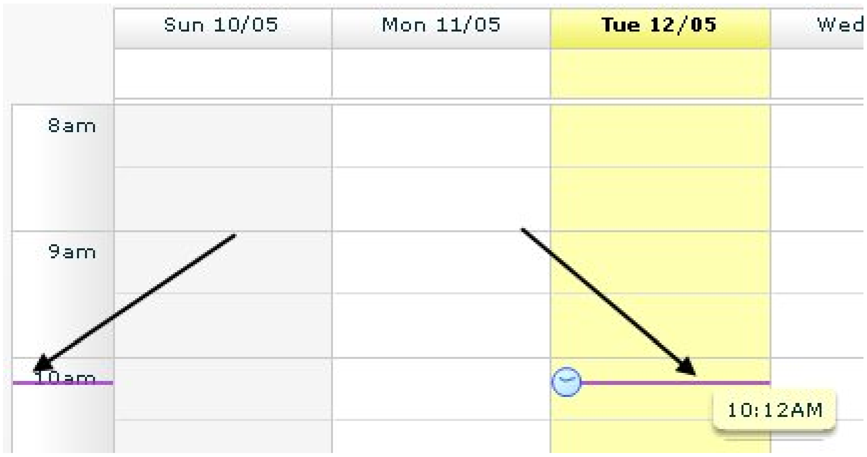
This method uses a default `itemColorFunction` that associates a color from the `itemColors` style property with each calendar. You can set the `itemColors` style property to specify different colors for each calendar. You can also provide a custom function for the `itemColorFunction` property to specify a color for each item of the data provider.

The following example uses a custom color function to display the events that end before 26th May 2008 in grey and the other events in green.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.calendar.CalendarItem;

      private function itemColorFunction(calItem:CalendarItem):Object {

        var color:uint;

        if (calItem.endTime < new Date(2008, 3, 16)) {
          color = 0xCCCCCC;
        } else {
          color = 0x00FF00;
        }
        return color;
      }
    ]]>
  </mx:Script>
  <ilog:Calendar id="cal" width="100%" height="100%"
                 startDate="{new Date(2008, 3, 13)}"
                 endDate="{new Date(2008, 3, 19)}"
                 itemColorFunction="{itemColorFunction}">
    <ilog:dataProvider>
      <mx:XMLList>
        <event startTime="2008/4/14 10:00" endTime="2008/4/14 16:00"
summary="Event1"/>
        <event startTime="2008/4/16 10:00" endTime="2008/4/16 16:00"
summary="Event2"/>
        <event startTime="2008/4/15 10:00" endTime="2008/4/18 16:00"
summary="Event3"/>
      </mx:XMLList>
    </ilog:dataProvider>
  </ilog:Calendar>
</mx:Application>
```

Current time indicator

The current time indicator represents the current time of day; it is displayed as a line in day or week mode. This indicator is not visible in month mode.You can hide this indicator by setting the `showCurrentTimeIndicator` property to false. You can customize the appearance of the line using the `currentTimeIndicatorStroke` property.



The following example shows a way to customize the current time indicator.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <ilog:Calendar width="100%" height="100%">
    <ilog:currentTimeIndicatorStroke>
      <mx:Stroke color="#0000FF" caps="none" weight="3" />
    </ilog:currentTimeIndicatorStroke>
  </ilog:Calendar>
</mx:Application>
```

Managing calendar events

Describes the management of events on calendars.

In this section

Calendar data events
Describes the events that occur on calendar data.

Navigation data events
Describes the events that occur when the user navigates in a calendar.

Time range selection events
Describes the events that occur when a time range selection is made.

Sheet events
Describes the events that occur when the user clicks in the sheet.

Event notification
Describes how a calendar notification event is dispatched to notify the user when an event in the calendar is due to start.

Calendar data events

A Calendar control fires events of several types in response to user interactions with the data. You can create an event handler for each of these event types.

Calendar data event	Description
change	Indicates that the selection changed as a result of user interaction.
itemClick	Indicates that the user clicked the pointer over a visual item in the control.
itemDoubleClick	Indicates that the user double-clicked the pointer over a visual item in the control.
itemRollOut	Indicates that the user rolled the pointer out of a visual item in the control.
itemRollOver	Indicates that the user rolled the pointer over a visual item in the control.
rightArrowClick	Indicates that user clicked the right arrow of a horizontal item renderer that is partially visible.
leftArrowClick	Indicates that user clicked the left arrow of a horizontal item renderer that is partially visible.

The calendar data events are of type `CalendarEvent`.

Navigation data events

The Calendar control fires an event of type `visibleTimeRangeChange` when the visible time range is modified, typically as a result of a user interaction.

The event `visibleTimeRangeChange` indicates that the visible time range has changed.

The calendar visible time range event is of type `CalendarEvent`.

The `startDate` and `endDate` properties of the event are the visible start and end days. The end date is included in the range.

For example, if the range is from 10th May to 13th May, four days are displayed.

The following example shows how to display and update the visible time range using a label.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="vertical">
  <mx:Script>
    <![CDATA[
      import ilog.calendar.CalendarEvent;
      private function visibleRangeChange(event:CalendarEvent):void {
        visibleTimeRange.text = event.startDate.toDateString() + " to " +
                                event.endDate.toDateString() + " included";
      }
    ]]>
  </mx:Script>
  <mx:HBox>
    <mx:Button label="prev" click="calendar.previousRange()" />
    <mx:Button label="next" click="calendar.nextRange()" />
    <mx:Label id="visibleTimeRange" />
  </mx:HBox>
  <ilog:Calendar id="calendar" width="100%" height="100%"
visibleTimeRangeChange="visibleRangeChange(event)" />
</mx:Application>
```

Time range selection events

The calendar control fires `timeRangeSelection` events when the user selects a time range on the calendar grid.

The `timeRangeSelection` events indicate that the user is selecting a time range.

Calendar time range event	Description
`timeRangeSelectionBegin`	Indicates that a time range selection has been initiated.
`timeRangeSelectionChange`	Indicates that the time range selection has changed. The `timeRangeSelection` property contains the current selection.
`timeRangeSelectionResize`	Indicates that the time range selection has been resized.
`timeRangeSelectionEnd`	Indicates that a time range selection has ended. The `reason` property of the event is set to `CalendarEventReason.COMPLETED` if the time range selection has successfully ended and to `CalendarEventReason.CANCELLED` otherwise.

The calendar time range selection events are of type `CalendarEvent`. For a range, the time of day in hours and minutes is taken into account as well as the date.

The following example shows how to display and update the time range selection with a label.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="vertical">
  <mx:Script>
    <![CDATA[
      import ilog.calendar.CalendarEvent;
      private function timeRangeSelectionChange(event:CalendarEvent):void {
        var selectedTimeRange:Array = calendar.timeRangeSelection;
        if (selectedTimeRange == null) {
          timeRangeSelection.text = "No selection";
        } else {
          timeRangeSelection.text = selectedTimeRange[0] + " to " +
selectedTimeRange[1];
        }
      }
    ]]>
  </mx:Script>
  <mx:Label id="timeRangeSelection" />
  <ilog:Calendar id="calendar" width="100%" height="100%"
                 timeRangeSelectionChange="timeRangeSelectionChange(event)" /
>
</mx:Application>
```

Sheet events

The calendar control fires an event when the user clicks in the column or row header or elsewhere in the sheet.

You can create an event handler for each of the following event types.

Calendar sheet event	Description
`sheetClick`	Indicates that the user clicked in the calendar.
`sheetDoubleClick`	Indicates that the user double-clicked in the calendar.

These events are of type `CalendarEvent`. The area clicked is available in the `calendarArea` property. The property `startDate` contains the time information related to the clicked area.

The calendar areas are described in the `CalendarArea` class.

In month mode:

- If the area is the row header (`CalendarArea.ROW_HEADER`), the `startDate` property contains the first day of the clicked week.
- If the area is the column header (`CalendarArea.COLUMN_HEADER`), the `startDate` property contains the first date for the specified day of the week in the displayed time range.

In the other modes:

- If the area is the row header (`CalendarArea.ROW_HEADER`), only the hour part of the `Date` object is meaningful.
- If the area is the column header (`CalendarArea.COLUMN_HEADER`), the `startDate` property contains the clicked date.

The `columnHeaderEnabled` or `rowHeaderEnabled` property must be set to `true` for the Calendar control to dispatch calendar events when the user clicks in the corresponding header.

Event notification

When an event represented by a data item in the calendar is about to start, an `eventNotification` calendar event is dispatched by the Calendar control. The `reminderInterval` property of the Calendar control specifies the time at which the notification event is dispatched as a number of minutes before the start time of the event. By default the reminder interval is 10 minutes.

User interaction with calendars

Item selection

When an item is clicked, it is selected and an `itemClick` event is fired. To enable multiple selection, set the `allowMultipleSelection` property to `true`.

The current selection is available in the `selectedItems` property of the calendar. You can disable selection by setting the `allowSelection` property to `false`. The selection actions are shown in the following table.

Action	Mouse
Select	Click an item in the calendar.
Extend a selection	Hold down the CTRL key and click the items.
Clear a selection	Click anywhere in the calendar background.

To cycle around the items using the keyboard, use the `CTRL + Left` or `CTRL + Right` keys.

Time range selection

Time range selection involves clicking the calendar grid and then dragging the selection. You can disable time range selection by setting the `allowTimeRangeSelection` property to `false`.

To select a time range using the keyboard, use the arrow keys to move a minimum time selection and Shift + an arrow key to extend the selection.

Event editing

Describes the event editing process and how to use calendar events associated with the editing of events when the user moves or resizes events in the calendar control by direct manipulation.

In this section

About event editing
Discusses event editing in general.

Defining tool tips
Describes the types of tool tips used and how to define them.

Event editing process
Gives the sequence of steps when an event is edited.

Using editing events
Describes the calendar editing events and how to use them to customize the editing of events in the calendar.

Accessing event data in an event listener
Describes the data accessible within an event listener.

Determining the kind of edit in an event listener
Describes how to determine the kind of edit operation from a Calendar property.

Determining the reason for an itemEditEnd event
Describes how to determine the reason for the end of an edit operation from a CalendarEvent property.

Examples of editing event handlers
Gives several examples demonstrating the use of editing events.

About event editing

Editing events in a Calendar has some similarities with editing cells in Flex list-based components. However there are also differences: there is no item editor when you edit events and there is no `itemEditBegin` calendar event; you receive intermediate calendar events for each change while editing an event.

To disable editing, set the `editable` property to `false`. The `moveEnabled` and `resizeEnabled` properties of the Calendar govern whether the user can move and resize the events respectively. The `moveEnabledFunction` and `resizeEnabledFunction` properties specify a custom function (if any) to control the editing capabilities for a specific event.

To represent the event being edited during editing, the Calendar uses either the default item renderers or the item renderers specified by the `itemVerticalRenderer`, `itemHorizontalRenderer`, and `itemLabelRenderer` properties.

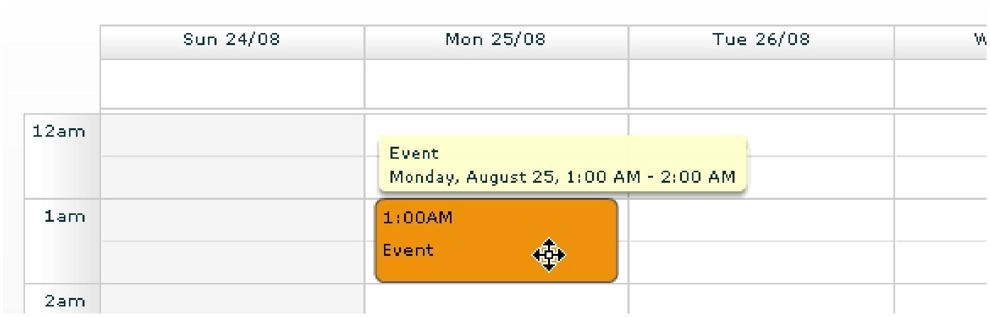
For more information on the item renderers, see *Calendar item renderers*.

Defining tool tips

Event data tool tips

The `Calendar` component uses two tool tips for a data item:

- ♦ A data tip displayed when the mouse pointer hovers over the event
- ♦ An editing tip displayed while the event is being moved or resized to provide feedback on the ongoing operation



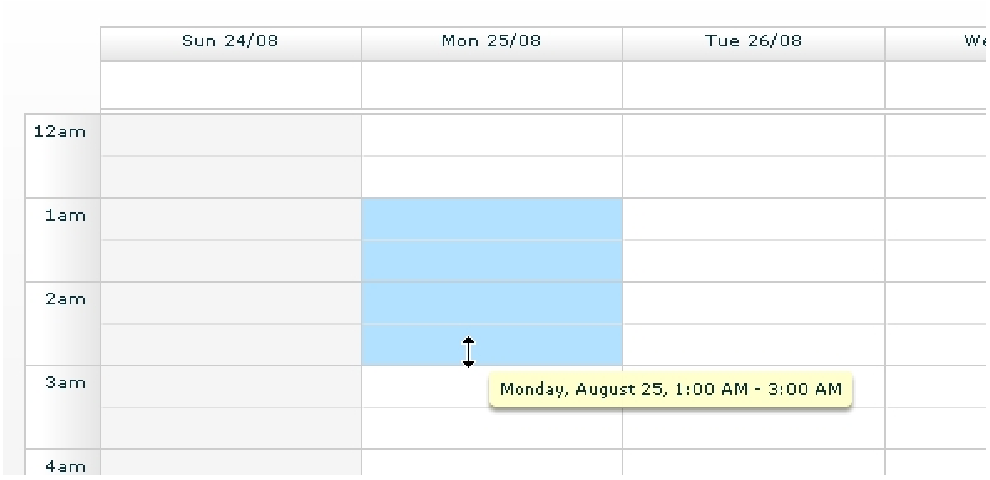
By default both tool tips show the same information: the summary of the event, its start and end time, and its duration. You can customize the contents of the data tip by setting the `dataTipFunction` property of the `Calendar` object. You can customize the editing tip by setting the `editingTipFunction` property of the Calendar object.

The following example shows how to use the `dataTipFunction` and `editingTipFunction` properties to specify custom tool tip texts.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import mx.utils.StringUtil;
      import mx.formatters.DateFormatter;
      import ilog.calendar.CalendarItem;
      private var dateFormatter:DateFormatter = new DateFormatter();
      private function initApp():void {
        dateFormatter.formatString = "YYYY/MM/DD JJ:NN";
      }
      public function formatTip(item:CalendarItem):String {
        return StringUtil.substitute("From {0} to {1}",
                          dateFormatter.format(item.startTime),
                          dateFormatter.format(item.endTime));
      }
    ]]>
  </mx:Script>
  <ilog:Calendar width="100%" height="100%" date="{new Date('2008/4/13')}"
dataTipFunction="{formatTip}" editingTipFunction="{formatTip}">
    <ilog:dataProvider>
      <mx:XMLList>
        <event startTime="2008/4/14 10:00" endTime="2008/4/14 16:00"
summary="Event1"/>
        <event startTime="2008/4/16 10:00" endTime="2008/4/16 16:00"
summary="Event2"/>
        <event startTime="2008/4/15 10:00" endTime="2008/4/18 16:00"
summary="Event3"/>
      </mx:XMLList>
    </ilog:dataProvider>
  </ilog:Calendar>
</mx:Application>
```

Time range selection tool tip

A Calendar component displays a data tip when a time range selection is being made.



You can customize this data tip by setting the `timeRangeTipFunction` property of the Calendar object.

The following example shows how to use the `timeRangeTipFunction` property to specify custom tool tip text.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import mx.utils.StringUtil;
      import mx.formatters.DateFormatter;
      import ilog.calendar.CalendarItem;
      private var dateFormatter:DateFormatter = new DateFormatter();
      private function initApp():void {
        dateFormatter.formatString = "YYYY/MM/DD JJ:NN";
      }
      public function formatTip(date1:Date, date2:Date):String {
        return StringUtil.substitute("From {0} to {1}",

                         dateFormatter.format(date1),
                         dateFormatter.format(date2));
      }
    ]]>
  </mx:Script>
  <ilog:Calendar width="100%" height="100%" date="{new Date('2008/4/13')}"
                 timeRangeTipFunction="{formatTip}" >
  </ilog:Calendar>
</mx:Application>
```

Event editing process

The following sequence of steps occurs when an event is edited:

1. The user holds down the mouse button on an event and starts dragging.

   The kind of editing operation performed depends on which area of the event is being dragged:

   - ♦ The start extremity – when dragged, resizes the event by modifying its start time.
   - ♦ The end extremity – when dragged, resizes the event by modifying its end time.
   - ♦ The body of the event – when dragged, moves the event.

2. The Calendar dispatches the `itemEditBegin` calendar event and shows the editing tool tip. You can use this calendar event to prepare data for editing.

   The next two steps can be repeated any number of times while the user is dragging.

   For details of the editing tool tip, see *Defining tool tips*

3. The user moves or resizes the event.

4. The Calendar dispatches the `itemEditMove` or `itemEditResize` calendar event. You can use these calendar events to modify the event that is being edited.

   For more information, see *Enforcing a minimum duration when resizing events*.

5. The user ends the editing session. Typically the event editing session ends when the user releases the mouse button.

6. The Calendar dispatches the `itemEditEnd` calendar event to update the data item and Calendar. You can use this calendar event to modify the event item or update the data item in a custom way.

   For more information, see *Updating the data provider item using a custom date format*.

7. The event that was being edited is updated in the Calendar.

Using editing events

A Calendar component dispatches the following calendar events as part of the event editing process: the `itemEditBegin`, `itemEditMove`, `itemEditResize`, and `itemEditEnd` calendar events. The Calendar control defines default event listeners for all of these calendar events except `itemEditBegin`.

You can write your own event listeners for one or more of these calendar events to customize the editing process. When you write your own event listener, it executes before the default event listener defined by the component; the default listener executes afterward.

You can also replace the default event listener for the component with your own event listener. To prevent the default event listener from executing, you call the `preventDefault()` method from anywhere in your event listener.

Use the following events when you want to customize the editing of events:

itemEditBegin

Dispatched when the editing session starts. The user has pressed the mouse button over an event and started dragging.

The `editKind` property is set with the editing operation being performed:

- `CalendarItemEditKind.MOVE_REASSIGN` – the user is dragging the event body
- `CalendarItemEditKind.RESIZE_START` – the user is dragging the start extremity of the event
- `CalendarItemEditKind.RESIZE_END` – the user is dragging the end extremity of the event

The Calendar component does not have a default listener for the `itemEditBegin` event. After the event is sent the Calendar displays the editing tool tip which will provide information on the event while it is being edited.

For details of the editing tool tip, see *Defining tool tips*

You can write an event listener for this event to modify the `CalendarItem` data used by the item renderer, or to modify some other information that will be used during editing.

For more information see *Accessing event data in an event listener*.

itemEditMove

Dispatched when the event is being moved. The Calendar component has a default listener for the `itemEditMove` event that adjusts the start and end time of the `CalendarItem`.

The default event listener performs the following actions:

- Rounds the start time according to the current view:
  - In Month mode, start time is rounded to the nearest start of day.
  - In Week or Day mode:
    - A start time for a short event is rounded to the time slot duration (see `getTimeSlotDuration` method)
    - A start time for a long event is rounded to the nearest start of day

- ♦ Updates the end time so the duration of the event remains constant.

You can write an event listener for this event to examine and modify the `CalendarItem` data used by the item renderer.

For more information see *Accessing event data in an event listener* and *Custom snapping when moving short events*.

Use the `isLongEvent` method to find out whether an event is long or short. For more information see *Calendar item renderers*.

itemEditResize

Dispatched when the event is being resized. The Calendar component has a default listener for the `itemEditResize` event that adjusts the start or end time of the `CalendarItem` object.

The default event listener performs the following actions:

- ♦ Rounds the start or end time, depending on the kind of edit operation and the current view:
  - ● In Month mode, the start or end time is rounded to the nearest start of day:
    - ♦ A short event cannot be resized.
    - ♦ A long event has a minimum duration of one day.
  - ● In Week or Day mode:

    A short event start or end time is rounded to the current time slot duration (see `getTimeSlotDuration()` method). A short event has a minimum duration equal to the duration of the current time slot.

    - ♦ A long event start or end time is rounded to the nearest start of day. A long event has a minimum duration of one day.
- ♦ Makes sure that the start time is before the end time and that the event has a minimum duration.

You can write an event listener for this event to modify the `CalendarItem` data used by the item renderer.

For more information see *Enforcing a minimum duration when resizing events*.

The following limitations apply:

- ♦ In month mode, a short item label renderer cannot be resized.
- ♦ In month mode, a long event cannot be resized into a short event.

itemEditEnd

Dispatched when the event editing session ends, typically when the user releases the mouse button.

The Calendar component has a default listener for this event that updates the item in the data provider with the values from the `CalendarItem` object associated with this item.

The default event listener performs the following actions:

- When the edit operation was not cancelled, the default event listener calls the `commitItem` function. This function uses the `startTimeField` and `endTimeField` properties to determine which properties of the data provider item must be updated and stores the edited start and end times in those fields. This behavior applies even if the `startTimeFunction` or `endTimeFunction` property is defined. In that case, you may want to specify a `commitItem` function that applies the changes to the edited data item.

When the data provider item is an XML object, the values of `startTime` and `endTime` are stored as Strings using the `Date.toString` method.

When the data provider item is an ActionScript® object, the values of the `startTime` and `endTime` are stored as `Date` objects.

You typically write an event listener for this calendar event to perform the following actions:

- Store the start and end times in a custom type inside the data provider. In this case you need to call `preventDefault()` to stop the Calendar from storing data using the fields defined on the Calendar. Whenever you store the start and end times with custom types, you must make sure that the calendar handles the types you use or provide your own functions specified with the `startTimeFunction` and `endTimeFunction` properties.

  For more information see *Updating the data provider item using a custom date format*.

- Store custom properties, other than the start time and end time identifier, that have been modified during editing.

After all listeners of `itemEditEnd` have been called, the Calendar component forces an update of the event by calling `ICollectionView.itemUpdated()` for this data provider item, to refresh the Calendar.

Accessing event data in an event listener

From within an event listener you have access to the data provider item and the current `startTime` and `endTime` of the event. To access the data provider item you use the `item` property of the event. To access the current `startTime` and `endTime` associated with the event, you use the `itemToCalendarItem` method using the data provider item as parameter. The resulting object is an instance of `CalendarItem` that wraps the data provider item and exposes the start and end times as `Date` instances.

The following example shows an event listener for the `itemEditEnd` event that accesses the data provider item and the calendar item to update the data provider item with the modified start and end times.

```
  private function onItemEditEnd(event:CalendarEvent):void
  {
    if (event.reason == CalendarEventReason.CANCELLED) {
      return;
    }

    event.preventDefault();

    var item:Object = event.item;
    var calendarItem:CalendarItem = Calendar(event.target).itemToCalendarItem
(item);

    item.startTime = calendarItem.startTime.toDateString();
    item.endTime = calendarItem.endTime.toDateString();

  }
```

In this example the listener for the `itemEditEnd` event updates the data provider item, setting the start and end times with the `String` representation containing only the date part (the time of day is not stored).

Determining the kind of edit in an event listener

When the user starts editing an event, the `Calendar` component determines the kind of edit operation from the item area where the user clicked. In the body of the event listeners for the `itemEditBegin`, `itemEditMove`, `itemEditResize`, and `itemEditEnd` events, you can determine the kind of edit and then handle it accordingly.

The `Calendar` class defines the `editKind` property, which contains a value that indicates the type of edit operation in progress. The `editKind` property has the values defined in the class `CalendarItemEditKind`.

Values of the editKind property

Value	Description
MOVE	Specifies that the user has clicked in the body of the event and is moving the event. You may get this value for `itemEditBegin`, `itemEditMove` and `itemEditEnd` events.
RESIZE_END	Specifies that the user has clicked the end extremity of the event and is resizing it by changing the end time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.
RESIZE_START	Specifies that the user has clicked the start extremity of the event and is resizing it by changing the start time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.

The following example shows how to retrieve the kind of edit within an event listener.

```
private function onItemEditBegin(event:CalendarEvent):void {
  var editKind:String = Calendar(event.target).editKind;
      //...
}
```

Determining the reason for an itemEditEnd event

A user can end an event editing operation in several ways. In the body of the event listener for the `itemEditEnd` event, you can determine the reason for the event, and then handle it accordingly. The `CalendarEvent` class defines the `reason` property, which contains a value that indicates the reason for the event. The `reason` property has the values defined in the class `CalendarEventReason`.

Value of the reason property

Value	Description
CANCELLED	Specifies that the user canceled editing and that they do not want to save the edited data.
COMPLETED	Specifies that the user completed the edit session, usually by releasing the mouse button.

Examples of editing event handlers

Enforcing a minimum duration when resizing events

In the following example, the listener for the `itemEditResize` event enforces the minimum duration of 1 hour for short events.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <mx:Script>
    <![CDATA[
      import ilog.calendar.CalendarItemEditKind;
      import ilog.calendar.CalendarItem;
      import ilog.calendar.CalendarEvent;

      private var isLongItemEdited:Boolean;

      private function onItemEditBegin(event:CalendarEvent):void {
        //store if the item is a long or a short item before the editing.
        isLongItemEdited = calendar.isLongEvent(event.item);
      }

      private function onItemEditResize(event:CalendarEvent):void {

        //cancel the default handler
        event.preventDefault();

        var calItem:CalendarItem = calendar.itemToCalendarItem(event.item);

        //round the edited date like the calendar does.
        if (calendar.editKind == CalendarItemEditKind.RESIZE_START) {
          calItem.startTime = calendar.roundDate(calItem.startTime,
isLongItemEdited);
        } else { // Resize end
          calItem.endTime = calendar.roundDate(calItem.endTime,
isLongItemEdited);
        }

        //ensure a minimum duration of 1 hour for a short event.
        calendar.ensureCalendarItemMinimumDuration(calItem, isLongItemEdited,
 60);

      }
    ]]>
  </mx:Script>
  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 itemEditBegin="onItemEditBegin(event)"
                 itemEditResize="onItemEditResize(event)"
                 date="{new Date(2008,0,15)}">
    <mx:XMLList>
      <event summary="Event 1" startTime="2008/01/14 08:00:00" endTime="2008/
01/14 17:00:00"/>
    </mx:XMLList>
  </ilog:Calendar>
</mx:Application>
```

Custom snapping when moving short events

The following example shows how to use the `itemEditMove` event to snap the start of a short event when moving it. In this example, the listener for the `itemEditMove` event snaps the start time of short events to the nearest hour.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <mx:Script>
    <![CDATA[
      import ilog.utils.TimeUnit;
      import ilog.calendar.CalendarItemEditKind;
      import ilog.calendar.CalendarItem;
      import ilog.calendar.CalendarEvent;

      private var isLongItemEdited:Boolean;

      private function onItemEditBegin(event:CalendarEvent):void {
        //store if the item is a long or a short item before the editing.
        isLongItemEdited = calendar.isLongEvent(event.item);
      }

      private function onItemEditMove(event:CalendarEvent):void {

        //cancel the defauld handler
        event.preventDefault();

        var calItem:CalendarItem = calendar.itemToCalendarItem(event.item);


        var duration:Number = calItem.endTime.time - calItem.startTime.time;

        //round short event with a snap duration of 1 hour

       var date:Date = calendar.roundDate(calItem.startTime, isLongItemEdited,
 60);

        //don't go too far
        if (isLongItemEdited && date > calendar.endDisplayedDate) {
          date = calendar.endDisplayedDate;
        }

        //adjust end time to keep the duration
        var date2:Date = new Date(date.time + duration);
        //don't go too far
        if (isLongItemEdited) {
          var date3:Date = calendar.calendar.addUnits(calendar.
startDisplayedDate, TimeUnit.DAY, 1);
          if (date2 < date3) {
            date2 = date3;
            date = new Date(date2.time - duration);
          }
        }

        calItem.startTime = date;
        calItem.endTime = date2;
      }
    ]]>
  </mx:Script>
  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 itemEditBegin="onItemEditBegin(event)"
                 itemEditMove="onItemEditMove(event)"
                 date="{new Date(2008,0,15)}">
    <mx:XMLList>
      <event summary="Event 1" startTime="2008/01/14 08:30:00" endTime="2008/
01/14 17:30:00"/>
    </mx:XMLList>
  </ilog:Calendar>
</mx:Application>
```

Updating the data provider item using a custom date format

The following example shows how to access dates stored with a custom date format. The start and end time are read from the data provider item using custom functions that parse the custom date format. A handler for the`itemEditEnd` event updates the data provider item after editing is complete using the custom date format for the start time and end time.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[

      import ilog.calendar.CalendarEvent;
      import ilog.calendar.CalendarItem;
      import ilog.calendar.CalendarEventReason;
      import mx.formatters.DateFormatter;

      private var replaceDashes:RegExp = new RegExp("-","g");
      private var replaceT:RegExp = new RegExp("T");
      private var iso8601Formatter:DateFormatter = new DateFormatter();

      private function initApp():void {
        iso8601Formatter.formatString = "YYYY-MM-DDTJJ:NN:SS";
      }

      private function readStartDate(item:Object):Date {
        return readDate(item, "@startTime");
      }

      private function readEndDate(item:Object):Date {
        return readDate(item, "@endTime");
      }

      private function readDate(item:Object, dateField:String):Date {
        var dateString:String = item[dateField].toString();

        dateString = dateString.replace(replaceDashes, "/");
        dateString = dateString.replace(replaceT, " ");

        return new Date(dateString);
      }

      // When setting startTimeFunction or endTimeFunction
      // you must provide a itemEditEnd handler to update the data
      // provider item. In this case we update the start and end
      // time in the data provider item using a ISO 8601 date
      // formatter.
      private function onItemEditEnd(event:CalendarEvent):void {

        //prevent the execution of the default handler
        event.preventDefault();

        if (event.reason == CalendarEventReason.CANCELLED) {
          return;
        }

       var calendarItem:CalendarItem = calendar.itemToCalendarItem(event.item)
;

       event.item.@startTime = iso8601Formatter.format(calendarItem.startTime)
;
        event.item.@endTime = iso8601Formatter.format(calendarItem.endTime);
      }


    ]]>
  </mx:Script>

  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 startTimeFunction="readStartDate"
                 endTimeFunction="readEndDate"
                 itemEditEnd="onItemEditEnd(event)"
                 date="{new Date(2008,0,15)}">
    <mx:XMLList>
      <event summary="Event 1" startTime="2008-01-14T08:00:00" endTime="2008
-01-15T17:00:00"/>
```

```
      <event summary="Event 2" startTime="2008-01-16T08:00:00" endTime="2008
-01-17T17:00:00"/>
      <event summary="Event 3" startTime="2008-01-18T18:00:00" endTime="2008
-01-21T17:00:00"/>
    </mx:XMLList>
  </ilog:Calendar>

</mx:Application>
```

In this example, the start time and end time of tasks are stored in the task data provider using the ISO8601:2000 extended format. Custom functions are set on the `startTimeFunction` and `endTimeFunction` properties of the calendar control. These functions parse the ISO8601:2000 formatted `String` and retrieve `Date` objects, stored in the `CalendarItem` object associated with the data provider item. The listener for the `itemEditEnd` event prevents the default update of the data provider item and instead uses a custom date formatter to store the start and end times as Strings in the IS8601:2000 format.

Recurring events

Explains how recurrence works and describes the support for recurring events.

In this section

Recurrence
Explains the recurrence concepts: recurring event, recurrence rule, recurrence engine.

Architecture of recurrence classes
Describes the classes used to support recurring events.

Data fields and custom functions
Describes the data fields and custom functions of the default recurrence descriptor and how they are used.

Recurrence rules
Describes the recurrence rule class, the engine that processes recurrence rules, and the definition of recurring events including recurrence rules.

Recurrence exceptions
Describes the types of recurrence exception and the definition of recurring events including recurrence rules.

Editing recurring events
Describes how to edit a recurring event, how to define exceptions to a recurrence rule, and how to prevent editing.

Recurrence

A recurring event is an event that is repeated several times according to a set of recurrence rules.

A recurring event is not directly displayed by the calendar; only the visible generated occurrences of this event are displayed.

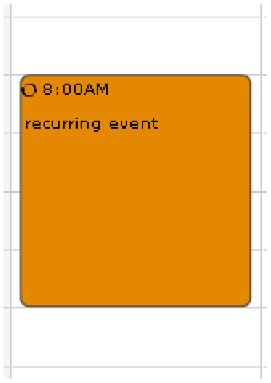
A recurrence rule defines a rule of repeating pattern or a recurring event.

The supported recurrence rule definition is based on the iCalendar (RFC-2445) specification.

Note: Only a subset of the syntax of a recurrence rule (RRULE) is used by the calendar. The rest of the iCalendar specification is *not supported*.

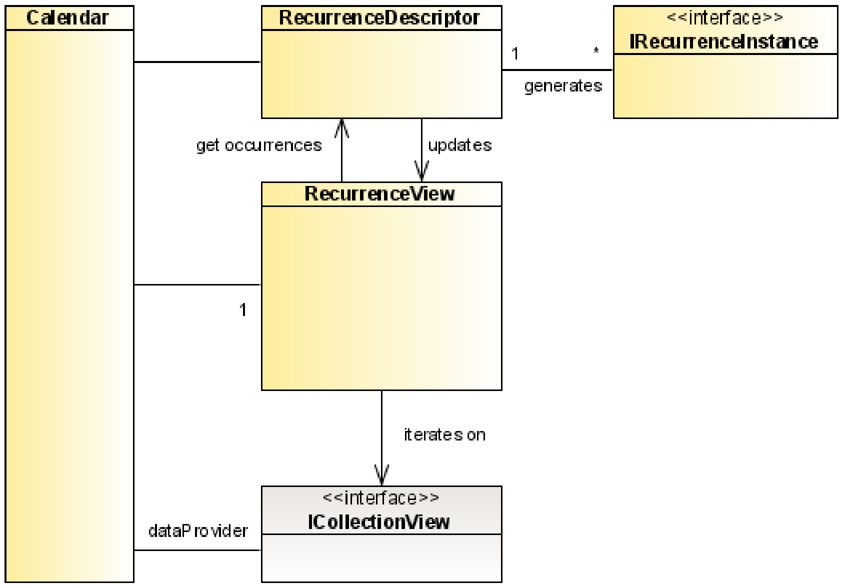
The Calendar provides a recurrence engine based on the implemented subset of the recurrence rules defined in the iCalendar specification (RFC-2445). This engine supports multiple inclusion and exclusion recurrence rules and dates.

A recurring event is displayed by the item renderers with a dedicated icon.



Architecture of recurrence classes

The architecture of the calendar classes for recurring events is shown in the following figure.



The `Calendar` component uses a `RecurrenceDescriptor` class that implements the `IRecurrenceDescriptor` interface to:

- ♦ Determine whether an item of the data provider is a recurring event.
- ♦ Generate the recurrence instances (or occurrences) of a recurring event in a specified time range.

The instance of `IRecurrenceDescriptor` is accessible through the `recurrenceDescriptor` property.

The recurrence view is then used by the `Calendar` to iterate over the nonrecurring events and the visible occurrences of the recurring events. The recurrence view is accessible through the `recurrenceView` property.

An occurrence of a recurring event is an instance of an implementation of `IRecurrenceInstance`. By default it is an instance of `RecurrenceInstance`.

An item that implements the `IRecurrenceInstance` interface is a generated occurrence of a recurring event. Generated occurrences are not data provider items.

Data fields and custom functions

The default recurrence descriptor supports various fields and functions.

Data fields

The field properties used to map the data that is used to render the recurring events are described in the following table. These fields are configurable on the recurrence descriptor.

Property	Default value	Purpose	Value format
recurringField	recurring or @recurring	The value of the field whose name is stored in this property is used to determine whether an item of the data provider is a recurring event.	A boolean or a string representing a Boolean
recurrenceRulesField	rrules or @rrules	The value of the field whose name is stored in this property is used to determine the recurrence rules for a recurring event.	Either a string containing a list of recurrence rules separated by vertical bar characters "|" or an array of RRule.
exceptionRecurrenceRulesField	exrules or @exrules	The value of the field whose name is stored in this property is used to determine the recurrence rules of occurrences to exclude from the recurrence set for a recurring event.	A list of recurrence rules as strings separated by a vertical bar character "|" or an array of RRule.
recurrenceDatesField	rdates or @rdates	The value of the field whose name is stored in this property is used to determine the dates to include in the recurrence set.	A comma-separated string that can be parsed by the Date.format() method or a custom function or an array of Date instances.
exceptionRecurrenceDatesField	exdates or @exdates	The value of the field whose name is stored in this property is used to determine the dates to exclude from the recurrence set.	A comma-separated string that can be parsed by the Date.format() method or a custom function or an array of Date instances.

The following example shows how to specify the field of the recurrence rules.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 date="{new Date(2008,0,15)}" >
    <mx:XMLList>
      <event summary="recurring event"
             startTime="2008/01/13 08:00:00"
             endTime="2008/01/13 10:00:00"
             recurrenceRule="FREQ=DAILY"
             recurring="true"/>
    </mx:XMLList>
    <ilog:recurrenceDescriptor>
      <ilog:RecurrenceDescriptor recurrenceRulesField="recurrenceRule" />
    </ilog:recurrenceDescriptor>
  </ilog:Calendar>
</mx:Application>
```

Custom functions

Custom functions can be used to retrieve the values from the data provider item. The following table shows the function property corresponding to each field property on `recurrenceDescriptor`.

Field property	Function property
`recurringField`	`recurringFunction`
`recurrenceRulesField`	`recurrenceRulesFunction`
`exceptionRecurrenceRulesField`	`exceptionRecurrenceRulesFunction`
`recurrenceDatesField`	`recurrenceDatesFunction`
`exceptionRecurrenceDatesField`	`exceptionRecurrenceDatesFunction`

The following example uses the `recurringFunction` to determine whether a data provider is a recurring event.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.calendar.recurrence.IRecurrenceInstance;
      import ilog.calendar.CalendarItem;
        private function isReccurringEvent(item:XML):Boolean {
        return item.@rrules.toString() != "";
        }
      ]]>
  </mx:Script>
  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 date="{new Date(2008,0,15)}" >
    <mx:XMLList>
      <event summary="Regular Event" startTime="2008/01/14 14:00:00"
endTime="2008/01/14 17:00:00"/>
      <event summary="Recurring event" startTime="2008/01/13 08:00:00"
```

```
endTime="2008/01/13 10:00:00" rrules="FREQ=DAILY" recurring="true"/>
    </mx:XMLList>
    <ilog:recurrenceDescriptor>
      <ilog:RecurrenceDescriptor recurringFunction="{isReccurringEvent}" />
    </ilog:recurrenceDescriptor>
  </ilog:Calendar>
</mx:Application>
```

Recurrence rules

Describes the recurrence rule class, the engine that processes recurrence rules, and the definition of recurring events including recurrence rules.

In this section

Recurrence rule class
Describes the recurrence rule class.

Recurrence engine
Describes the recurrence engine and how it differs from the specification.

Examples of recurring events
Gives various examples of recurring events.

Recurrence rule class

A recurrence rule is represented by the `RRule` class or a string that conforms to the `iCalendar` specification.

The `RRule` class can parse or write an `iCalendar` recurrence rule (`toString()` method) .

A recurrence rule is separated into the following main parts:

- The frequency of the event: every year, month, week, or day (mandatory)
- The end of the range of the recurrence: after n occurrences or until a specific date
- A list of additional parameters to configure the recurrence: a list of months, days of the month, week numbers, week days, and so on

Recurrence engine

The implementation of the recurrence engine differs from the recurrence rule specification regarding the implementation of dates and weeks and support for rules and filters.

Implementation of dates and weeks

The date format used in the `UNTIL` parameter (see `until`) or in the included/excluded dates is not the date format defined by the iCalendar specification:

- ♦ The supported date formats are those of the `Date.parse()` method. Custom functions to read and wite dates can be specified on the recurrence descriptor.
- ♦ The `WKST` parameter is not supported; the engine uses the `firstDayOfWeek` property instead. An instance of this calendar is available on the `calendar` property of the `Calendar` component, see `calendar`. By default this value is set to `0` (Sunday); in the iCalendar specification, the default is `1` (Monday)
- ♦ In the iCalendar specification, the first week of the year has at least four days; in the implementation provided this is configurable as `GregorianCalendar.minimalDaysInFirstWeek` (see `minimalDaysInFirstWeek`. By default, this value is set to `1`.

Support for rules and filters

The recurrence engine does not support some frequency rules and filters:

- ♦ The frequency (FREQ) values HOURLY, MINUTELY, SECONDLY are not supported.
- ♦ The parameters BYHOUR, BYMINUTE, BYSECOND are not supported.
- ♦ The BYSETPOS parameter is not supported.

In addition, in the iCalendar specification, the `DTSTART` parameter of a `VEVENT` is used by the recurrence rule to determine the start of the recurrence set and to infer missing values (for example the start time in hours and minutes of the occurrences); the recurrence engine implemented uses the `startTime` field of a data item for these purposes.

The following table shows the supported recurrence rule parameters.

iCalendar parameter	Valid values
FREQ	"YEARLY", "MONTHLY", "WEEKLY", "DAILY"
UNTIL	A Date
COUNT	A positive integer
INTERVAL	A positive integer
BYDAY	A list of week days
BYMONTHDAY	A list of days in the month (positive from the start of the month or negative from the end of the month)
BYYEARDAY	A list of days in the year (positive from the start of the year or negative from the end of the year)
BYWEEKNO	A list of week numbers (positive from the start of the year or negative from the end of the year)
BYMONTH	A list of month numbers (positive)

The iCalendar specification

For more information on the `iCalendar` specification see *http://www.ietf.org/rfc/rfc2445.txt*. Section 4.3.10 describes a recurrence rule. Section 4.8.5.4 provides examples of recurrence rules.

Examples of recurring events

The following XML excerpts represent recurring events according to the recurrence rule syntax presented.

The following example shows two ways to represent a birthday.

```
<event summary="Birthday"
        startTime="2008/01/14"
        endTime="2008/01/15"
        rrules="FREQ=YEARLY"
        recurring="true"/>


 <event summary="Birthday"
        startTime="2008/01/14"
        endTime="2008/01/15"
        rrules="FREQ=YEARLY;BYMONTH=1;BYMONTHDAY=14"
        recurring="true"/>
```

The following example shows a meeting that takes place every Monday starting on the 7th January 2008 from 3:00 pm to 4:00 pm.

```
<event summary="Meeting"
        startTime="2008/01/07 15:00"
        endTime="2008/01/07 16:00"
        rrules="FREQ=WEEKLY;BYDAY=MO"
        recurring="true"/>
```

The following example shows an event occurring the first Monday and last Monday of every month starting on the 14th January 2008.

```
<event summary="Event"
        startTime="2008/01/14"
        endTime="2008/01/15"
        rrules="FREQ=MONTHLY;BYDAY=1MO,-1MO"
        recurring="true"/>
```

The following example shows a daily event that has two exceptions on the 14th and 17th of October.

```
<event startTime="Sun Oct 12 12:00:00 GMT+0200 2008"
        endTime="Sun Oct 12 13:30:00 GMT+0200 2008"
        summary="Recurring Event"
        rrules="FREQ=DAILY"
        recurring="true"
       exdates="Tue Oct 14 12:00:00 GMT+0200 2008,Fri Oct 17 12:00:00 GMT+0200
2008"/>
```

The following example shows an event that occurs every Wednesday and also Friday 17th October at a different time.

```
<event startTime="Sun Oct 15 12:00:00 GMT+0200 2008"
        endTime="Sun Oct 15 13:30:00 GMT+0200 2008"
        summary="Recurring Event"
        rrules="FREQ=WEEKLY;BYDAY=WE"
        recurring="true"
        rdates="Tue Oct 17 17:00:00 GMT+0200 2008"/>
```

The following event shows a recurring event that has two recurrence rules. This event occurs every Wednesday and the last Friday of the month.

```
<event startTime="Sun Oct 12 12:00:00 GMT+0200 2008"
        endTime="Sun Oct 12 13:30:00 GMT+0200 2008"
        summary="Recurring Event"
        rrules="FREQ=WEEKLY;BYDAY=WE|FREQ=MONTHLY;BYDAY=-FR"
        recurring="true" />
```

The following event shows a recurring event that occurs every Friday except any Friday 13th.

```
<event startTime="Sun Oct 12 12:00:00 GMT+0200 2008"
        endTime="Sun Oct 12 13:30:00 GMT+0200 2008"
        summary="Recurring Event"
        rrules="FREQ=WEEKLY;BYDAY=FR"
        exrules="FREQ=MONTHLY;BYMONTHDAY=13"
        recurring="true" >/
```

Recurrence exceptions

Describes the types of recurrence exception and the definition of recurring events including recurrence rules.

In this section

Types of recurrence exception
Lists the types of recurrence exception that may arise.

Exception events
Explains what an exception event is like, with an example.

Types of recurrence exception

A recurring event can have the following types of exception:

- A list of exclusion recurrence rules. All occurrences generated by these rules are removed from the recurrence set.

  These exclusion rules are defined on the recurring event. See *Data fields and custom functions*.

- A list of dates. These dates are removed from the recurrence set.

  These exclusion dates are defined on the recurring event. See *Data fields and custom functions*.

- A list of exception events.

Exception events

An exception event is an event defined in the data provider that replaces one (and only one) occurrence of a recurring event. An exception must not itself be a recurring event.

In order to make the association between the recurring event and its exception events, the exception events in the data provider must share an identifier which is defined in the field `idField` (or by the function specified in the field `idFunction`).

The following code excerpt shows an XML model with a recurring event and an exception event.

```
<event id="0"
        startTime="Mon Oct 13 11:30:00 GMT+0200 2008"
        endTime="Mon Oct 13 13:00:00 GMT+0200 2008"
        summary="Recurring event"
        rrules="FREQ=DAILY"
        recurring="true"/>

<event id="0"
        startTime="Thu Oct 16 12:30:00 GMT+0200 2008"
        endTime="Thu Oct 16 14:00:00 GMT+0200 2008"
        summary="Exception event"
        exdate="Thu Oct 16 11:30:00 GMT+0200 2008"/>
```

In this example the two events have the same value in the `id` field; this field contains the default value of the event identifier.

Important: This notion of identifier is not the same as the notion of `uid` in Flex (see `UIDUtil`). These events have the same `id` but must have a different `uid`.

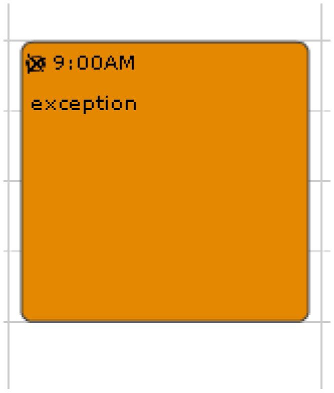
The exception event also defines the start time of the occurrence to replace. The field that contains this date is defined in the field `exceptionDateField` or by the function specified in the field `exceptionDateFunction`. By default the field `exdate` is used to retrieve this date.

In the example in this topic, the occurrence that should have started on October 16th at 11:30 is replaced by the exception event.

The association between a recurring event and its exception events can be customized by specifying a custom function in the property `getExceptionEventsFunction`. The custom function will return the list of exception events of a specified recurring event; it replaces the default implementation.

Note: In iCalendar, an exception can have THISANDPRIOR or THISANDFUTURE keywords to replace every occurrence before or after the exception. This is not supported by the Calendar component but it can easily be expressed as two recurring events: one recurring event has an end date (UNTIL) and the other recurring event starts at the next occurrence date.

An exception event is displayed by the item renderers with a dedicated icon.



Editing recurring events

Describes how to edit a recurring event, how to define exceptions to a recurrence rule, and how to prevent editing.

In this section

Editing a recurring event
Refers to the information on event editing in general and gives specific considerations for editing a recurring event.

Managing exceptions
Explains how to manage exception dates in order to handle exceptions to a recurrence rule.

Preventing the editing of a recurring event

Editing a recurring event

For more information on event editing in general see *Event editing*.

When an occurrence of a recurring event is edited (moved or resized) using the mouse, the default ITEM_EDIT_END handler applies the modification to the recurring event, thus modifying the entire series of occurrences.

For example, the data provider contains a recurring event that repeats every day at 10:00 am. An occurrence is edited and moved to 9:00am. By default, all the occurrences are moved to start at 9:00am.

Managing exceptions

After the editing of a recurring event, the exception dates defined on this recurring event and the associated exception events must be updated accordingly.

Recurrence exception dates

The exception dates listed by default in the `exdates` field (See *Data fields and custom functions*) are updated after an editing gesture by the `updateExceptionDates` method.

This method is called by the `applyItemEditEnd` method, itself called by the default `ITEM_EDIT_END` handler of the Calendar component.

This operation can be customized by specifying a custom function in the `updateExceptionDatesFunction` property. This is mandatory, for example, if you specify a custom function in `exceptionRecurrenceDatesFunction` to get the exception dates by a function instead of using a field.

If you are preventing the default `ITEM_EDIT_END` handler and are not using the `applyItemEditEnd` method, do not forget to call the `updateExceptionDates` method to manage the exception dates update.

Exception events

After the editing of a recurring event, the exception events must also be updated. In fact, an exception event defines the start time of the occurrence to be replaced. As the recurring event has changed, this occurrence start time is no longer valid.

Each exception event of an edited recurring event is updated by the `updateExceptionEventDate` method.

This method is called by the `applyItemEditEnd` method of the Calendar component, itself called by the default `ITEM_EDIT_END` handler of the Calendar.

This operation can be customized by specifying a custom function in the `updateExceptionEventDateFunction` property of the `RecurrenceDescriptor` instance. This is mandatory, for example, if you specify a custom function in the `exceptionDateFunction` property of the `RecurrenceDescriptor` instance to get the exception dates by function instead of using a field.

If you are preventing the default `ITEM_EDIT_END` handler and are not using the `applyItemEditEnd` method, do not forget to call the `updateExceptionEventDate` method to manage the exception events update.

For more information on event editing in general see *Event editing*.

If the association between a recurring event and its exception events changes (add, remove) the `invalidateExceptionEvents` method must be called to recompute the association between the recurring events and its exceptions.

Preventing the editing of a recurring event

To prevent the editing of a recurring event, set the `moveEnabledFunction` and `resizeEnabledFunction` as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

  <mx:Script>
    <![CDATA[
      import ilog.calendar.recurrence.IRecurrenceInstance;

      private function moveResizeEnabled(item:Object):Boolean {
        return !(item is IRecurrenceInstance);
      }
    ]]>
  </mx:Script>
  <ilog:Calendar id="calendar"
                 width="100%" height="100%"
                 moveEnabledFunction="{moveResizeEnabled}"
                 resizeEnabledFunction="{moveResizeEnabled}"
                 date="{new Date(2008,0,15)}">
    <mx:XMLList>
      <event summary="Regular Event" startTime="2008/01/14 14:00:00"
endTime="2008/01/14 17:00:00"/>
      <event summary="recurring event" startTime="2008/01/13 08:00:00"
endTime="2008/01/13 10:00:00"
                 rrules="FREQ=DAILY" recurring="true"/>
    </mx:XMLList>
  </ilog:Calendar>
</mx:Application>
```

Working with Date objects

IBM ILOG Elixir provides a set of utility classes to help manipulate `Date` objects:

- `TimeUnit` defines time units
- `GregorianCalendar` provides operations for accurately manipulating `Date` objects

The calendar control uses an instance of `GregorianCalendar` that you can retrieve from the `Calendar.calendar` property. It is important to use the same instance of `GregorianCalendar` that is used by the calendar control to achieve consistent results.

The `GregorianCalendar` class provides the following methods:

- `addUnits`, allows the adding of time units to a `Date` object or the removing of time units from a `Date` object.
- `floor`, gets the floor of a `Date` object for a given time unit.
- `getWeek`, returns the number of the week for a `Date` object.
- `round`, rounds a `Date` object to the nearest time unit, for a given time unit.

For complete reference information, see the `TimeUnit` class and the `GregorianCalendar` class.

For more information on `Date`, see *Working with dates and times* in *Adobe Programming ActionScript 3.0*.

Gantt Charts

Describes how to use IBM ILOG Elixir Gantt charts with Adobe® Flex® 3.

In this section

Resource charts
Describes how to use IBM ILOG Elixir resource charts with Adobe® Flex® 3.

Task charts
Describes how to use IBM ILOG Elixir task charts with Adobe® Flex® 3.

Resource charts

Describes how to use IBM ILOG Elixir resource charts with Adobe® Flex® 3.

In this section

Introduction to resource charts
Describes resource charts and gives an example.

Resource chart architecture
Describes the architecture of the IBM ILOG Elixir resource chart classes and their relationship to the Adobe® Flex® 3 types..

Creating a resource chart control
Describes how to define a resource chart control.

Configuring resource chart data
Describes how to configure resource chart data.

Configuring a resource chart control
Describes how to configure a resource chart control.

Resource chart item renderers
Describes the default task item renderer used to display the graphic representation of a task data item in the Gantt sheet, explains how the task item renderer gets the information it needs to display a task, and describes the use of custom renderers in the Gantt sheet and in the data grid.

Styling a resource chart
Describes how to style the data grid, Gantt sheet, task items, and time scale in a resource chart.

Animating a resource chart
Describes how to use the built-in animations in a resource chart.

Managing resource chart events
Describes how to manage events that are a result of user interactions with a resource chart.

User interaction with resource charts
Describes possible user interactions with resource charts.

Editing tasks in a resource chart
Describes the task editing process and how to use events associated with task editing when the user moves, resizes or reassigns tasks in the resource chart control by direct manipulation.

Working with Date objects
Describes how to use the utility classes for manipulating Date objects.

Introduction to resource charts

A resource chart is a Gantt chart used to display the allocation or scheduling of resources. It is typically found in planning and production scheduling applications.

The `ResourceChart` control is a list of resources that display their properties in columns and their associated tasks over a time line. Resources may be organized as a hierarchy, either to match a hierarchy of resources or to arrange them into groups or categories.

The resource chart control is a composite control composed of:

- ♦ A data grid that displays the hierarchy of resources and their properties. It uses the Adobe® Flex® `AdvancedDataGrid` control.

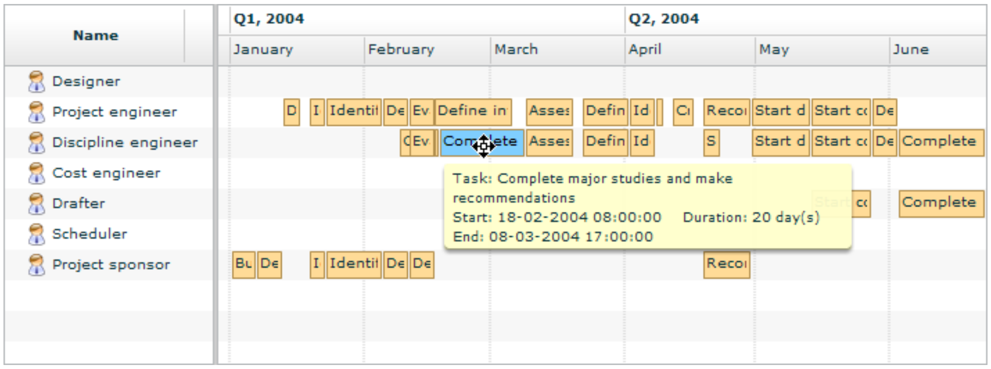
  You are assumed to be familiar with using the `AdvancedDataGrid` control and working with data providers. For more information on the `AdvancedDataGrid` control, see *Using the AdvancedDataGrid control* in *Adobe Flex 3 Developer's Guide*.

- ♦ A Gantt sheet that displays the tasks arranged in time. Tasks are laid out in rows corresponding to the resources they are assigned to. The Gantt sheet header displays a time scale that supports navigatiing and zooming on the time axis.

The Gantt sheet displays from back to front:

- ♦ A row grid to help visualize the rows
- ♦ A work grid that renders the working and nonworking times
- ♦ A time grid to help visualize the time units
- ♦ The tasks
- ♦ A time indicator to show the current time

The following is an example of a resource chart.



Task: Complete major studies and make recommendations
Start: 18-02-2004 08:00:00 Duration: 20 day(s)
End: 08-03-2004 17:00:00

The resource chart is characterized by the ability to:

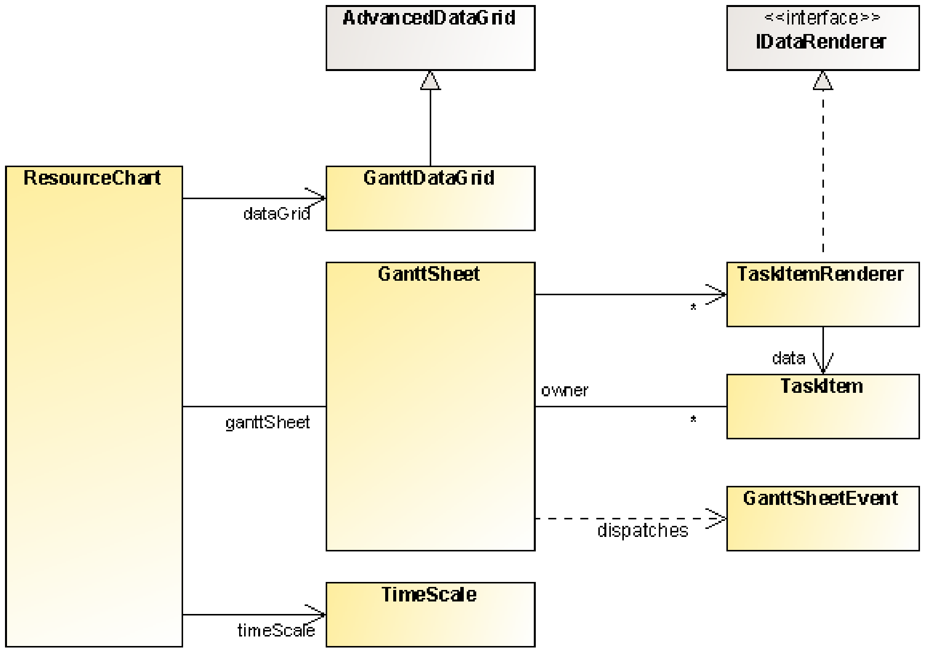
- ♦ Display resources as a hierarchy in a data grid by leveraging the features of the Adobe® Flex® `AdvancedDataGrid`.

- Display tasks over a time line inside a Gantt sheet.
- Edit task duration, start, and end times in response to direct manipulation with the mouse.
- Reassign tasks to resources in response to direct manipulation with the mouse.
- Dispatch events related to hovering over, clicking, double-clicking, and editing tasks.
- Navigate along the time axis and focus on time periods.

A task corresponds to some work to be completed. A task is assigned to a resource. A milestone is a special kind of task of zero duration that corresponds to an event.

Resource chart architecture

The resource chart architecture is shown in the following figure.



The resource chart classes shown in the UML diagram are described in the following table.

Main resource chart classes

Class	Description
`ResourceChart`	The resource chart component. It extends the `UIComponent` class. It is the top-level object to be used in MXML.
`GanttDataGrid`	The component that displays the data grid for the resources. The GanttDataGrid class extends the `AdvancedDataGrid` class.
`GanttSheet`	The component that displays the Gantt sheet. It extends the `UIComponent` class.
`GanttSheetEvent`	Defines the type of events dispatched by the Gantt sheet.
`TimeScale`	The component that displays the time scale. It extends the `UIComponent` class.
`TaskItemRenderer`	The default item renderer for tasks in the Gantt sheet.
`TaskItem`	The data that is set on the task item renderers.

Creating a resource chart control

Defining a resource chart control in MXML

You can define a `ResourceChart` control in MXML using the `<ilog:ResourceChart>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value.

A `ResourceChart` control displays resource and task data and can be configured by using two data providers as shown in the following example.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.core.Container;
      import mx.containers.DividedBox;
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}"/>
</mx:Application>
```

Configuring resource chart data

A resource chart control gets its data from two data providers: a resource data provider containing the resource data items, and a task data provider containing the task data items.

The resource data provider can be a flat list or can contain hierarchical or grouped resource data. It is set using the `resourceDataProvider` property of the `ResourceChart` object. The data grid of the resource chart leverages all features of the `AdvancedDataGrid` it is built on, and most considerations regarding the definition of data for an `AdvancedDataGrid` object apply to the resource data provider. For more information see *Hierarchical and grouped data display* in *Using the AdvancedDataGrid control* in *Adobe Flex 3 Developer's Guide*.

> **Note**: You must set the resource data provider using the `resourceDataProvider` property on the `ResourceChart` instance. Setting a data provider on the data grid will have no effect.

The task data provider is a flat list of tasks. It is set using the `taskDataProvider` property of the ResourceChart object. For more information see *The task data provider*.

In addition to specifying the resource data provider and the task data provider of the resource chart control, you must also:

- ♦ Specify the fields of the resource items that are used as the columns in the data grid. For more details see *Specifying resource columns*.
- ♦ Specify the fields of the task items that are used to represent the tasks in the Gantt sheet. For more details see *Specifying task data items with field mappings* and *Specifying task data items with a custom function*.
- ♦ Specify the fields of data provider items that are used to associate the tasks with the resources. For more detail see *Specifying the relationship between resources and tasks*.

The task data provider

The task data provider is a list-based data provider. For more information on list-based data providers see *Using Data Providers and Collections* in the *Adobe Flex 3 Developer's Guide*.

You specify the task data for the `ResourceChart` control by using the `taskDataProvider` property.

The task data provider supports the following types of data:

XML
A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag, or an `XML` or `XMLList` object.

IList or ICollectionView
An object that implements the `IList` or `ICollectionView` interface (such as an instance of the `ArrayCollection` or `XMLListCollection` class), whose data provider conforms to the structure specified in either of these items.

Other objects
An array of items or an object.

When setting the `taskDataProvider` property the `ResourceChart` control modifies its internal representation of the task data as follows:

- ♦ XML or XMLList are wrapped in an instance of `XMLListCollection`.
- ♦ `IList` or `ICollectionView` are used directly.
- ♦ An `Array` object is wrapped in an instance of `ArrayCollection`.
- ♦ An ActionScript® object of any other data type is wrapped in an `ArrayCollection` using an `Array` containing the object as its sole entry.

For example, you pass an `Array` to the `taskDataProvider` property. If you read the data back from the `taskDataProvider` property it is returned as an instance of `ArrayCollection`.

The best practice when your task data can change dynamically is to use a collection, which provides the necessary notifications of changes.

Specifying resource columns

To specify the resource properties that are shown on the Gantt data grid, map the properties of a resource to columns. This mapping follows the scheme defined by `AdvancedDataGrid`.

The following example shows how to do this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:dataGrid>
```

```
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
          <mx:AdvancedDataGridColumn dataField="location"
                                     headerText="Location"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:ResourceChart>
</mx:Application>
```

This example displays the Name and Location columns in the Gantt data grid, mapped from the `name` and `location` properties of resource items.

See *Specifying columns* in *Creating a DataGrid control* in *Adobe Flex 3 Developer's Guide* for more information.

Specifying task data items with field mappings

In addition to specifying the `taskDataProvider` of the `ResourceChart` control, to display (or render) the tasks you must map the fields of task data items (start time, end time, and label) to fields that exist in the task data provider by setting field properties in the control. In the task data provider, a field is either a property (if the task data provider items are ActionScript objects) or an attribute (if the task data provider items are XML objects).

The following table shows the field properties in a `ResourceChart` control that contain the field names used to map task data. In each case, the first default value of the field name is for an ActionScript object and the second (with @ prefix) is for an XML object.

Field properties for the ResourceChart control

Property	Default value	Purpose
`taskEndTimeField`	`endTime` or `@endTime`	The value of the field whose name is stored in this property is used as the end time of each task data item.
`taskLabelField`	`name` or `@name`	The value of the field whose name is stored in this property is used to display the label of each task data item.
`taskStartTimeField`	`startTime` or `@startTime`	The value of the field whose name is stored in this property is used as the start time of each task data item.
`taskIsMilestoneField`	`milestone`	The value of the field whose name is stored in this property is used to determine whether a task is a milestone.

The value of fields `taskEndTimeField` and `taskStartTimeField` must be either a `Date` object or any `String` representation of a date that is supported by the constructor of `Date` objects.

The value of the field `taskIsMilestoneField` must be either a Boolean object or a String with the possible values `true` or `false`.

Specifying task data items with a custom function

If you need an alternative way of retrieving the values of task data items, you can use the `taskEndTimeFunction`, `taskLabelFunction`, `taskStartTimeFunction`, and `taskIsMilestoneFunction` properties of the control to specify custom functions that will be used to obtain the end time, label, and start time and the milestone flag respectively of each task data item from the `taskDataProvider` object.

The following example shows how you can use your own date-parsing function to read the start and end time in XML.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
    <mx:Script>
      <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);

        public function parseDate(value:String):Date {
          return new Date(value);
        }
        public function startTimeFunction(item:Object):Date {
          return parseDate(item.startTime);
        }
        public function endTimeFunction(item:Object):Date {
          return parseDate(item.endTime);
        }
      ]]>
    </mx:Script>

    <ilog:ResourceChart width="100%" height="100%"
                        resourceDataProvider="{resources}"
                        taskDataProvider="{tasks}"
                        taskStartTimeFunction="{startTimeFunction}"
                        taskEndTimeFunction="{endTimeFunction}"/>
</mx:Application>
```

In this example the parseData function mimics the default behavior.

You can use the taskLabelFunction property to specify a function that returns a label depending on the item in a similar way.

Specifying the relationship between resources and tasks

The resource chart control associates resources and tasks as follows:

- ♦ One resource can have zero or more tasks
- ♦ One task can have zero resources or one resource

This association is built using fields of the data provider items which are specified with properties of the ResourceChart class as shown in the following table.

Field properties in the resource-task association

Property	Default value	Description
resourceIdField	id or @id	A uniquer identifier of the resource within the resource data provider. This is a field of the resource data provider items.
taskResourceIdField	resourceId or @resourceId	The identifier of the resource associated with the task. This is a field of the task data provider items.

If changes are made to the relationship outside the resource chart control, you must notify the task data provider for the tasks concerned by calling its ICollectionView.itemUpdated() method.

The ResourceChart class provides the following methods to access the association:

- ♦ getResource(task:Object):Object – returns the resource associated with a task
- ♦ getTasks(resource:Object):Array– returns the tasks associated with a resource

For complete information on the properties and the associated methods see the ResourceChart class.

Configuring a resource chart control

Describes how to configure a resource chart control.

In this section

Hiding resource columns
Describes how to hide resource columns in a resource chart control.

Specifying resource icons in the data grid
Describes how to change the resource icons used in a resource chart control.

Specifying the visible time range
Describes how to specify the visible time range in a resource chart control.

Specifying the minimum and maximum visible time
Describes how to set the minimum and maximum visible time in a resource chart control.

Specifying the minimum and maximum zoom factors
Describes how to set the minimum and maximum zoom factors in a resource chart control.

Defining the tooltips for tasks
Describes the different tooltips for tasks and how to define them.

Defining the working and nonworking periods
Describes how to customize the working and nonworking periods.

Snapping tasks when moving or resizing
Describes how to snap tasks when moving or resizing them.

Hiding resource columns

To hide a column you can set its `visible` property to `false` as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
          <mx:AdvancedDataGridColumn dataField="location"
                                     headerText="Location"
                                     visible="false"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:ResourceChart>
</mx:Application>
```

In this example, you display only the `Name` column in the Gantt data grid, which is mapped from the `name` property of a resource. The `Location` column is no longer displayed.

See *Hiding and displaying columns* in *Creating a DataGrid control* in *Adobe Flex 3 Developer's Guide* for more information.

Specifying resource icons in the data grid

You can change the icons used by the Gantt data grid in different ways, since it leverages the icon-customization support provided by the `AdvancedDataGrid` class.

The following example shows how to change the icons used for branches (Team resources) and the icons used for leaves (Member resources).

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var resourcesCollection:ArrayCollection =
        new ArrayCollection([
          { id: "T1", name: "Geneva Team", type: "team", children: [
            { id: "R1", name: "Project Manager",
              location: "Geneva", type: "member" },
            { id: "R2", name: "Developer",
              location: "Geneva", type: "member" }]
          }
        ]);

      [Bindable]
      public var resources:HierarchicalData =
        new HierarchicalData(resourcesCollection);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection ([
        { id: "T1", resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { id: "T2", resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { id: "T3", resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" },
        { id: "T4", resourceId: "R2", name: "Implement project",
          startTime: "1/21/2008 17:0:0", endTime: "1/30/2008 17:0:0" }
      ]);

      [Embed(source="team.png")]
      public var Team:Class;

      [Embed(source="member.png")]
      public var Member:Class;

      public function customIconFunction(item:Object):Class {
        return (item.type == "team") ? Team : Member;
      }
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid iconFunction="customIconFunction">
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:ResourceChart>
</mx:Application>
```

This example shows how the `iconFunction` property of the `AdvancedDataGrid` object is used to set a custom function, `updateIconFunction`, which checks the `type` property of the processed item. In the example data, the value for the `type` property is `team` for a Team resource and `member` for a Member resource.

See *Hierarchical and grouped data display* for the `AdvancedDataGrid` control in *Adobe Flex 3 Developer's Guide* for more information on the various options for customizing the icons in an `AdvancedDataGrid` component.

Specifying the visible time range

The visible time range defines the range of time that is displayed in the Gantt sheet. You can specify the visible time range with the `visibleTimeRangeStart` and the `visibleTimeRangeEnd` properties of the Gantt sheet.

The following example shows how to set the visible time range from 1 January 2007 00:00:00 to 31 December 2007 24:00:00.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2007 8:0:0", endTime: "1/15/2007 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2007 8:0:0", endTime: "1/17/2007 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2007 8:0:0", endTime: "1/21/2007 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="600" height="400"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet
          visibleTimeRangeStart="{new Date(2007, 0, 1)}"
          visibleTimeRangeEnd="{new Date(2007, 11, 31, 24)}" />
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

When the visible time range is not set explicitly, the resource chart displays the range of time corresponding to the data provided.

Specifying the minimum and maximum visible time

The minimum and maximum visible time define the navigable time range. The visible time range will always be within this navigable time range. Depending on your application, it can be convenient to limit the navigable time range to prevent end users from being lost in time. A typical read-only resource chart should limit navigation to the range of available data.

The following example shows you how to set the minimum and maximum visible time from 1 January 2007 00:00:00 to 31 December 2010 24:00:00.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet minVisibleTime="{new Date(2007, 0, 1)}"
                       maxVisibleTime="{new Date(2010, 11, 31, 24)}"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

The minimum visible time should be on or after 1 January 100.

The maximum visible time should be on or before 1 January 10000.

Specifying the minimum and maximum zoom factors

The minimum and maximum zoom factors define the range of possible values of the zoom factor in the Gantt sheet. The zoom factor, expressed in milliseconds per pixel, defines the correspondence from time to screen space and conversely from screen space to time. It corresponds to the precision that users can obtain along the time axis.

For example, in an application where the precision of the start and end of tasks is shown in days, you can limit the space allocated to represent a day to a maximum of 30 pixels. The following example shows how to limit the space in this way.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.utils.TimeUnit;
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet
        minZoomFactor="{TimeUnit.DAY.milliseconds / 30}"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example, the maximum zoom factor is not explicitly set; it is left unbounded.

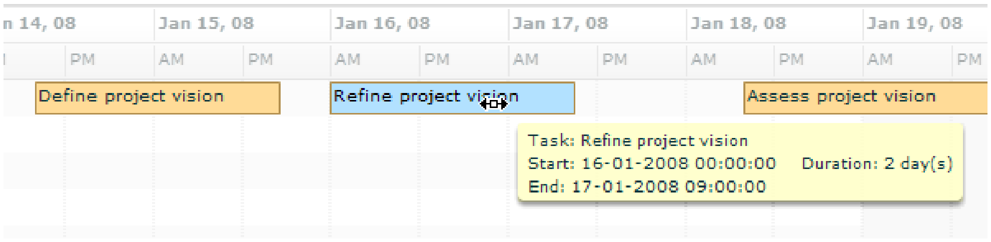
The minimum zoom factor should be greater than `TimeUnit.MILLISECONDS.milliseconds / 1`.

The maximum zoom factor should be less than `TimeUnit.DECADE.milliseconds / 20`.

Defining the tooltips for tasks

The Gantt sheet component uses two tooltips for tasks: a data tip displayed when the mouse pointer hovers over the task and an editing tip displayed while tasks are being moved or resized to provide feedback on the ongoing operation.

The following image shows the default tooltip for a task.



Task: Refine project vision
Start: 16-01-2008 00:00:00 Duration: 2 day(s)
End: 17-01-2008 09:00:00

By default both tooltips show the same information: the name of the task, its start and end time and its duration. You can customize the contents of the data tip by setting the `dataTipField` or the `dataTipFunction` properties of the `GanttSheet` object. You can customize the editing tips by setting the `editingTipFunction` property of the `GanttSheet` object.

The following example shows how to use the `dataTipFunction` and `editingTipFunction` properties to specify custom tooltip texts.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;
      import ilog.gantt.ConstraintKind;
      import ilog.core.DataItem;
      import ilog.gantt.ConstraintItem;
      import ilog.gantt.GanttSheet;
      import ilog.gantt.TaskItem;

      import mx.collections.ArrayCollection;
      import mx.formatters.DateFormatter;
      import mx.utils.StringUtil;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Project",
          startTime: "1/14/2008 8:0:0", endTime: "2/5/2008 17:0:0",
          children: [
            { id: "T2", name: "Inception",
              startTime: "1/14/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
            { id: "T3", name: "Elaboration",
              startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" },
            { id: "T4", name: "Construction",
              startTime: "1/22/2008 8:0:0", endTime: "2/1/2008 17:0:0",
              children: [
                { id: "T5", name: "Iteration 1",
                  startTime: "1/22/2008 8:0:0", endTime: "1/25/2008 17:0:0" }
,
                { id: "T6", name: "Iteration 2",
                  startTime: "1/26/2008 8:0:0", endTime: "1/29/2008 17:0:0" }
,
                { id: "T7", name: "Iteration 3",
                  startTime: "1/30/2008 8:0:0", endTime: "2/1/2008 17:0:0" },

              ]},
            { id: "T8", name: "Transition",
              startTime: "2/2/2008 8:0:0", endTime: "2/5/2008 17:0:0" },
          ]}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T2", toId:"T3", kind:"endToStart" },
        { fromId:"T3", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T8", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" },
        { fromId:"T6", toId:"T7", kind:"endToStart" }
      ]);

      private var dateFormatter:DateFormatter = new DateFormatter();

      private function initApp():void {
        dateFormatter.formatString = "YYYY/MM/DD JJ:NN";
      }

      public function formatTip(item:DataItem):String {
        if (item is TaskItem)
          return formatTaskTip(TaskItem(item));
        else if (item is ConstraintItem)
          return formatConstraintTip(ConstraintItem(item));

        // Should not happen
        return null;
      }

      private function formatTaskTip(item:TaskItem):String {
        return StringUtil.substitute("{0}\n{1} - {2}",
                         item.label,
                         dateFormatter.format(item.startTime),
                         dateFormatter.format(item.endTime));
      }

      private function formatConstraintTip(item:ConstraintItem):String {
        var fromExtremity:String;
        var toExtremity:String;
```

```
        switch (item.kind) {
          case ConstraintKind.END_TO_END:
            fromExtremity = "end";
            toExtremity = "end";
            break;
          case ConstraintKind.END_TO_START:
            fromExtremity = "end";
            toExtremity = "start";
            break;
          case ConstraintKind.START_TO_END:
            fromExtremity = "start";
            toExtremity = "end";
            break;
          case ConstraintKind.START_TO_START:
            fromExtremity = "start";
            toExtremity = "start";
            break;
        }

        var fromTaskItem:TaskItem = taskChart.ganttSheet.itemToTaskItem(item.
fromTask);
        var toTaskItem:TaskItem = taskChart.ganttSheet.itemToTaskItem(item.
toTask);

        return StringUtil.substitute("From {0} of: {1}\nTo {2} of: {3}",
                         fromExtremity,
                         fromTaskItem.label,
                         toExtremity,
                         toTaskItem.label);
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  creationComplete="taskChart.dataGrid.expandAll();"
                  >
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
    <ilog:ganttSheet>
      <ilog:GanttSheet dataTipFunction="formatTip"
                       editingTipFunction="formatTip"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example, the texts for the data tip and the editing tip are created using the same function; this function returns a text which provides the name of the task, the name of its resource, and the start and end time of the task in a custom format.

Defining the working and nonworking periods

The Gantt sheet control displays a work grid behind the tasks. The work grid shows the working and nonworking periods. The level of detail on the working and nonworking periods is dynamically adjusted as the zoom factor along the time axis changes.

The working and nonworking periods are defined by the following properties of the `GanttSheet` object.

Working and nonworking period properties

Property	Description
`nonWorkingDays`	The list of days of the week that are nonworking days.
`workingTimes`	The ranges of working time for each day of the week. Each day of the week can have none, one or several ranges of working time.
`nonWorkingRanges`	Additional arbitrary ranges of nonworking time. These ranges are defined as full days or ranges of time of any duration. The nonworking time periods defined by the `nonWorkingRanges` property take precedence over the working and nonworking times defined by the other properties.

For complete reference information, see the `GanttSheet` class.

For information on the rendering of the working and nonworking periods see *Work grid properties*.

Snapping tasks when moving or resizing

The Gantt sheet snaps the start and end time of tasks while they are being moved or resized. By default the Gantt sheet dynamically selects the snapping time precision based on the current zoom factor along the time axis.

You can define a constant snapping time precision by setting the `snappingTimePrecision` property of the Gantt sheet.

In the following example in MXML, the start and end times of tasks are snapped to a 1 day precision.

```
<ilog:GanttSheet snappingTimePrecision=”{{unit: TimeUnit.DAY, steps: 1}}”/>
```

You can also dynamically set the snapping time precision depending on the zoom factor along the time axis, by listening to the changes in the visible time range and adjusting the `snappingTimePrecision` property.

Resource chart item renderers

Describes the default task item renderer used to display the graphic representation of a task data item in the Gantt sheet, explains how the task item renderer gets the information it needs to display a task, and describes the use of custom renderers in the Gantt sheet and in the data grid.

In this section

Default task item renderer in resource chart
Describes the default task item renderer.

Task item renderer architecture in resource chart
Describes the task item renderer architecture.

Task item renderer layout in resource chart
Describes the layout of a task given by the task item renderer.

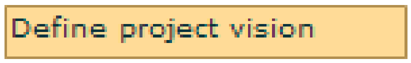
Defining a custom task item renderer for a resource chart
Explains how to define a custom task item renderer.

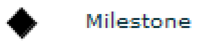
Defining a custom renderer for the data grid of a resource chart
Describes how to use custom renderers in the Gantt data grid of a resource chart.

Default task item renderer in resource chart

In the Gantt sheet, the default task item renderer displays a bar with a label and optional start and end symbols.

The following figures show examples of the default task item renderer for a task and a milestone.





The default task item renderer is defined by the `TaskItemRenderer` class.

You can customize the appearance of the task item renderer using styles and skins. For more information, see *Task styling for a task chart*.

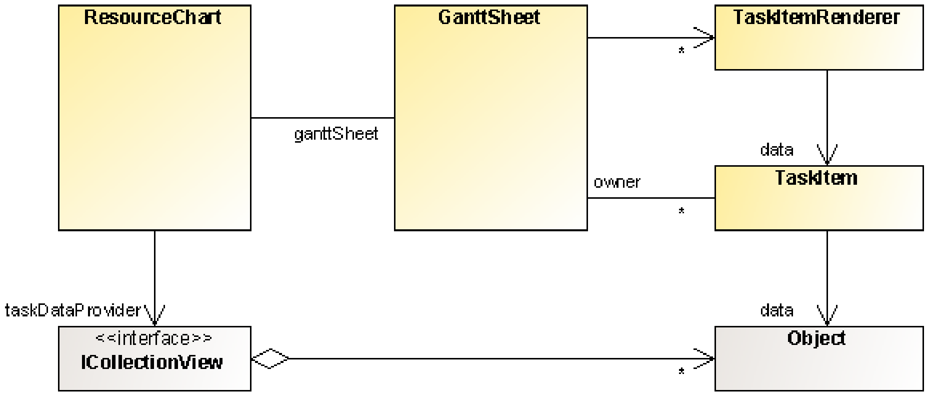
Task item renderer architecture in resource chart

The Gantt sheet component uses task item renderers to display the tasks from the task data provider. The same task item renderer is also used to display a task while it is being edited.

For a task item renderer to work with the Gantt sheet component, the Gantt sheet component must be able to pass information to the task item renderer. During editing, the task item renderer must also be able to pass updated information back to the Gantt sheet component.

As with Flex® list-based controls, the Gantt sheet component and the task item renderer use the `data` property to pass information. The contents of the `data` property for a task item renderer are a `TaskItem` object.

The following diagram shows the task item renderer and `TaskItem` object in context.



The `TaskItem` object has `startTime`, `endTime`, `label`, `resourceId` and `resource` properties that are initialized with the values of the corresponding fields in the data provider item, using the settings defined on the resource chart control. It also has a `data` property that references the data provider item, and an `owner` property that references its `GanttSheet` object.

The Gantt sheet component uses the `startTime` and `endTime` properties of the `TaskItem` object to determine the position and size of the renderer.

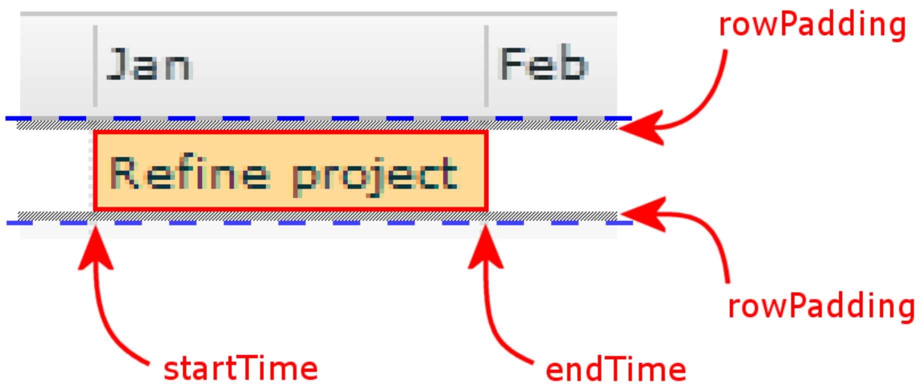
During editing the Gantt sheet component modifies the `startTime`, `endTime`, `resourceId` and `resource` properties of the `TaskItem` object. The `TaskItem` object keeps the `resource` and `resourceId` properties synchronized, so you can modify whichever suits you best; the other one is updated accordingly. The data provider item is updated only at the end of the editing process when the editing is successful.

For more information on the configuration of the `startTime`, `endTime`, `label`, and `resourceId` fields see *Configuring resource chart data*.

For more information on task editing, see *Editing tasks in a resource chart*.

Task item renderer layout in resource chart

The following figure shows how the position and size of a task item renderer relates to its containing row, and to the start and end time.



The red rectangle represents the bounds of the task item renderer.

You can also see that:

- ♦ The task item renderer is vertically padded inside the row according to the `rowPadding` style property of the `GanttSheet`.
- ♦ On the horizontal axis, the bounds of the task item renderer are defined by the start and end time of the task.
- ♦ The start and end symbols are drawn beyond the bounds of the task item renderer.

Defining a custom task item renderer for a resource chart

To display tasks from the task data provider, the Gantt sheet component uses either the default `TaskItemRenderer` class or the task item renderer specified by the `taskItemRenderer` property.

A custom task item renderer must:

- ♦ Extend the `DisplayObject` class or one of its subclasses. In practice the preferred base classes are `UIComponent`, or `Sprite` for a lighter component.
- ♦ Implement the `IDataRenderer` interface.
- ♦ Implement either the `IInvalidating` interface or the `IProgrammaticSkin` interface.

A custom task item renderer should:

- ♦ Implement the `ISimpleStyleClient` interface in order to use styling
- ♦ Implement the `ilog.gantt.IConstraintConnectionBounds` interface when the renderer can draw beyond its bounds which are defined by the `x`, `y`, `height` and `width` properties.

The x-coordinate and the width of the task item renderer are set by the Gantt sheet depending on the start and end time of the data item. The y-coordinate and the height of the task item renderer are set by the Gantt sheet based on the position and height of the row corresponding to the resource associated with the task which is defined in the data grid of the resource chart.

The contents of the `data` property for a task item renderer are a `TaskItem` object. The `TaskItem` class allows you to access the `startTime`, `endTime`, `label` and `isMilestone` fields, the task data provider item, the associated resource data provider item, the `GanttSheet` object, and the interaction state of the task item renderer.

You can access the `startTime`, `endTime`, `label` and `isMilestone` fields inside the task item renderer as follows.

```
 var taskItem:TaskItem = data as TaskItem;
  var startTime:Date = taskItem.startTime;
  var endTime:Date = taskItem.endTime;
  var label:String = taskItem.label;
  var isMilestone:Boolean = taskItem.isMilestone;
```

You can access the task data provider item from the task item renderer as follows.

```
  var taskItem:TaskItem = data as TaskItem;
  var dataProviderItem:Object = taskItem.data;
```

You can access the resource data provider item from the task item renderer as follows:

```
  var taskItem:TaskItem = data as TaskItem;
  var resource:Object = taskItem.resource;
```

Once you have the data provider item and the `TaskItem` object, you can retrieve the GanttSheet and access the interaction state of the task item renderer as follows.

```
var ganttSheet:GanttSheet = taskItem.owner as GanttSheet;
var isHighlighted:Boolean = ganttSheet.isItemHighlighted(dataProviderItem);
var isSelected:Boolean = ganttSheet.isItemSelected(dataProviderItem);
```

The following example in MXML shows how to declare an inline custom task item renderer that displays a label in a box whose background color depends on the end date of the task.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet>
        <ilog:taskItemRenderer>
          <mx:Component>
            <mx:Canvas horizontalScrollPolicy="off" clipContent="true"
                       backgroundColor="{getBackgroundColor(data)}">
              <mx:Script>
                <![CDATA[
                  private function getBackgroundColor(data:Object):uint
                  {
                    var color:uint;
                    if (data.endTime > new Date(2008, 0, 20))
                      color = 0xff5050;
                    else
                      color = 0xd0f0f0;
                    return color;
                  }
                ]]>
              </mx:Script>
              <mx:Label text="{data.label}" />
            </mx:Canvas>
          </mx:Component>
        </ilog:taskItemRenderer>
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example the task is rendered as a label in a box. The background of the task is red if its end time exceeds 2008-01-20. Otherwise the task has a light emerald background. You can drag the tasks to different positions to see changes in color.

Defining a custom renderer for the data grid of a resource chart

As a subclass of `AdvanceDataGrid`, the Gantt data grid leverages the custom renderer functionality provided by its base class.

See *Using item renderers with the AdvancedDataGrid control* in *Adobe Flex 3 Developer’s Guide* for information on creating custom renderers for the `AdvancedDataGrid`.

Styling a resource chart

Describes how to style the data grid, Gantt sheet, task items, and time scale in a resource chart.

In this section

Styles for resource charts
Describes the specific styles available for task rendering in a resource chart and for the inner components: Gantt sheet, task items, and time scale.

Data grid styling
Describes the styles available for data grid rendering.

Gantt sheet styling
Describes the style properties that allow you to customize the visual appearance of the Gantt sheet.

Time scale styling
Describes the style properties that allow you to customize the visual appearance of the time scale.

Styling of Gantt sheet items
Describes the style properties that allow you to customize the visual appearance of the items on the Gantt sheet.

Task styling for a resource chart
Describes the ways that you can style tasks in a resource chart when these are rendered using the default task item renderer.

Styles for resource charts

In addition to the styles provided by its base types, the resource chart provides a set of styles that can be used to customize the rendering of its inner components and tasks:

- ♦ `dataGridStyleName` – the name of the style used by the Gantt data grid
- ♦ `ganttSheetStyleName` – the name of the style used by the Gantt sheet
- ♦ `milestoneItemStyleName` – the name of the style used by the task item renderers for milestones
- ♦ `taskItemStyleName` – the name of the style used by the task item renderers for tasks
- ♦ `timeScaleStyleName` – the name of the style used by the time scale

By default the style names for task item renderers depends on whether the task item is a milestone or not. You can override this behavior by providing a custom function in the Gantt sheet property `itemStyleNameFunction`.The following example in MXML shows how you can customize these styles.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myGanttSheetStyle {
      workingColor : blue;
    }
    .myDataGridStyle {
      rollOverColor : red;
    }
    .myTimeScaleStyle {
      rollOverColor : red;
    }
    .myTaskItemStyle {
      rollOverColor : red;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}"
                      ganttSheetStyleName="myGanttSheetStyle"
                      dataGridStyleName="myDataGridStyle"
                      timeScaleStyleName="myTimeScaleStyle"
                      taskItemStyleName="myTaskItemStyle"/>
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the renderers of tasks and the inner components. The CSS classes customize the rollover color of the data grid, time scale and task item renderers; and the color of the work grid in the Gantt sheet.

See the `ResourceChart` class for more information on these styles.

Data grid styling

The Gantt data grid leverages all the built-in styles provided by `AdvancedDataGrid`, so you can use and set all the styles in the same way as you would with `AdvancedDataGrid`.

See the `AdvancedDataGrid` API in *Adobe Flex 3 Language Reference* for more information on the available styles.

Gantt sheet styling

In addition to the styles provided by its base types the Gantt sheet control defines its own styles.

The Gantt sheet display is composed of the following elements, arranged from back to front:

♦ Row grid

♦ Work grid

♦ Time grid

♦ Tasks – styling of the tasks is described in the section *Task styling for a task chart*

♦ Time indicator

Note: Keep this back-to-front order in mind when defining the color settings of your Gantt sheet. An opaque element in one layer hides the corresponding area in the underlying layers.

Row grid properties

The row grid is a set of horizontal visual elements that aid in associating tasks displayed in the Gantt sheet with their respective resources. The following are the main elements of the row grid, arranged from back to front:

♦ Rows where tasks are displayed. These rows are aligned with the resources corresponding to the tasks in the Gantt data grid. You can customize their styling with the following property: `alternatingItemColors`.

♦ Horizontal grid lines that delimit the rows. You can customize their styling with the following property: `horizontalGridLineColor`.

For the horizontal grid lines to be shown, you have to set the `showHorizontalGridLines` property on the Gantt sheet to `true`, as it is not displayed by default.

The following example shows how to customize the row grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);
      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet alternatingItemColors="{[0xECF6F4, 0xEDE5F3]}"
                       showHorizontalGridLines="true"
                       horizontalGridLineColor="0xFFFFFF" />
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the alternating item colors of the Gantt sheet and the color of the horizontal grid lines.

Work grid properties

The work grid shows vertical areas that represent the working and nonworking time ranges in the Gantt sheet. You can customize the work grid with the following properties: `nonWorkingAlpha`, `nonWorkingColor`, `nonWorkingFill`, `workingColor`, `workingAlpha`, `workingFill`.

If both `workingColor` and `workingFill` are set, `workingFill` takes precedence. By default, `workingFill` is not set, so `workingColor` is used. The same applies to `nonWorkingColor` and `nonWorkingFill`.

The following example shows how to customize the simpler work grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>
  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet workingColor="green" workingAlpha="0.5"
                       nonWorkingColor="yellow" nonWorkingAlpha="0.5"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the working and nonworking color and transparency of the work grid in the Gantt sheet.

The work grid can also be customized to use fill styles to paint the working and nonworking vertical areas. You can customize the fill styles in two ways:

1. In ActionScript® , retrieve the instance of `GanttSheet` and set its style properties with the appropriate instances of `IFill`.
2. In MXML, set the `ganttSheet` property of `ResourceChart` with a `GanttSheet` instance that has been customized with your fill styles.

The following example shows the MXML way.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet>
        <ilog:workingFill>
          <mx:LinearGradient>
            <mx:entries>
              <mx:GradientEntry color="red" ratio="0" alpha="0.5"/>
              <mx:GradientEntry color="blue" ratio="0.33" alpha="0.5"/>
            </mx:entries>
          </mx:LinearGradient>
        </ilog:workingFill>
        <ilog:nonWorkingFill>
          <mx:SolidColor color="yellow" alpha="0.4"/>
        </ilog:nonWorkingFill>
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the working and nonworking fill styles used to paint the corresponding areas of the Gantt sheet.

Time grid properties

The time grid shows vertical lines in the Gantt sheet that are aligned with the ticks in the minor row of the time scale. You can customize the styling of the time grid with the following properties: `timeGridAlpha` and `timeGridColor`.

The following example shows how to customize the time grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet timeGridColor="green" timeGridAlpha="0.9"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the color and transparency of the time grid in the Gantt sheet.

Time indicator properties

The current time indicator shows a vertical line corresponding to the current time in the Gantt sheet. You can customize the styling of the time indicator with the following properties: `currentTimeIndicatorAlpha` and `currentTimeIndicatorColor`.

For the current time indicator to be shown, you have to set the `showCurrentTimeIndicator` property on the Gantt sheet to `true`, as it is not displayed by default.

The following example shows how to customize the time indicator styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import ilog.utils.TimeUnit;
      import ilog.utils.GregorianCalendar;
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);

      [Bindable]
      private var start:Date;

      [Bindable]
      private var end:Date;

      private function initApp():void {
        // Make sure the current time is visible
        var calendar:GregorianCalendar = ganttSheet.calendar;
        var now:Date = new Date();
        var start:Date = calendar.addUnits(now, TimeUnit.WEEK, -1);
        var end:Date = calendar.addUnits(now, TimeUnit.WEEK, 1);
        ganttSheet.visibleTimeRangeStart = start;
        ganttSheet.visibleTimeRangeEnd = end;
      }
    ]]>
  </mx:Script>
  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                     resourceDataProvider="{resources}"
                     taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet id="ganttSheet"
                       showCurrentTimeIndicator="true"
                       currentTimeIndicatorColor="blue"
                       currentTimeIndicatorAlpha="0.8"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the color and transparency of the current time indicator in the Gantt sheet.

Animation properties

The animation properties are used to control the animation aspects of the Gantt sheet, namely:

- The animation duration that determines how long the animations will last
- The function used to determine the pace of progression of an animation

For more details on the available animation features, see *Animating a resource chart*.

The following example shows how to customize the animation-related styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.effects.easing.Linear;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet animationDuration="2000"
                       easingFunction="{Linear.easeNone}"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the Gantt sheet’s animation duration and the easing function.

To disable time navigation animation altogether, just set the `animationDuration` style of the Gantt sheet to zero as follows:

```
     <ilog:GanttSheet animationDuration="0"/>
```

Time scale styling

In addition to the styles provided by its base types the time scale control defines its own styles.

The time scale display is composed of the following elements:

- ♦ Background
- ♦ The time scale rows and separator
- ♦ Interaction feedback

There are two time scale rows: the major time scale row at the top and the minor time scale row at the bottom. Each time scale row is composed of ticks and text that do not overlap.

See also the `TimeScale` API for more information on the available styles.

Background properties

The background is rectangle that is drawn behind all elements of the time scale. Its style can be customized with the following property: `backgroundColors`.

The `backgroundColors` property takes an array with two colors. If the colors provided are the same, the background is painted with the specified color. Otherwise, the background is painted using a gradient that goes from the first color at the top of the time scale to the second color at the bottom.

The following example shows how to customize the background styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>
  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale backgroundColors="{['white', 'black']}"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the background colors of the time scale.

Separator properties

The separator is displayed by the time scale between the major and minor time scale rows. Its style can be customized with the following properties: `separatorAlpha`, `separatorColor` and `separatorThickness`.

The following example shows how you can easily customize the separator styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale separatorAlpha="0.5"
                      separatorColor="red"
                      separatorThickness="2"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the color, transparency and thickness of the separator of the time scale.

Tick properties

The ticks separate the time segments in both the major and minor time scale rows. You can customize the styling of ticks with the following properties: `majorTickAlpha`, `majorTickColor`, `minorTickColor` and `minorTickAlpha`.

The following example shows how to customize the tick styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale majorTickAlpha="0.5" majorTickColor="red"
                      minorTickAlpha="0.5" minorTickColor="blue"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the time scale’s major and minor ticks color and transparency.

Text properties

The text contains a label for each time interval shown by the time scale rows in the time scale. You can customize the styling of text with the following properties: `majorTextStyleName` and `minorTextStyleName`.

The following example shows how to customize the text styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myMajorTextStyle {
      color: blue;
      fontWeight: bold;
    }
    .myMinorTextStyle {
      color: red;
      fontStyle: italic;
    }
  </mx:Style>
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale majorTextStyleName="myMajorTextStyle"
                      minorTextStyleName="myMinorTextStyle"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the text of the major and minor rows in the time scale. The CSS classes customize the color, font style and weight of the text.

Highlighting properties

Highlighting comprises visual elements that aid in providing feedback to users who are interacting with the time scale. The following are the main highlighting visual elements:

♦ Highlighting of predefined time intervals when users hover over them on the time scale

♦ Highlighting of arbitrary time intervals when users select them on the time scale

The highlighting style can be customized with the following properties: `rollOverAlpha` and `rollOverColor`.

The following example shows how to customize the highlighting styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>
  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale rollOverAlpha="0.5" rollOverColor="red"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the rollover color and transparency of the time scale.

Styling of Gantt sheet items

The Gantt sheet assigns a style to task item renderers depending on the kind of task that is represented. The following table lists the kinds of tasks, the style name assigned by default, and the `TaskChart` property that defines the value.

Kind of item	Default style name	Property
milestone task	`milestoneTask`	`milestoneItemStyleName`
task	`task`	`taskItemStyleName`

You can provide a custom function to override the default mapping from the kind of object to the style name, using the `GanttSheet.itemStyleNameFunction`.

For more information on styling tasks, see *Task styling for a task chart*.

Task styling for a resource chart

The task display for the default task item renderer is composed of the following elements:

- Bar
- Start symbol
- End symbol
- Text
- Roll over and selection feedback

The `TaskItemRenderer` uses skins as shown in the following table.

Property name	Default value	Description
`barSkin`	`ilog.gantt.TaskBarSkin`	The skin used to render the task bar.
`startSymbolSkin`	`ilog.gantt.TaskSymbolSkin`	The skin used to render the start symbol of a task.
`endSymbolSkin`	`ilog.gantt.TaskSymbolSkin`	The skin used to render the end symbol of a task.

See the `TaskItemRenderer` API for more information on the available style properties.

Bar properties

The bar is a rectangle that is drawn from the start time to the end time of the task. The top and bottom margins allow you to position the bar vertically within the task item renderer. The margins are defined by the properties `barTopMargin` and `barBottomMargin`.

The colors of the bar and border are defined by the `backgroundColor` and `borderColor` properties. The width of the border is defined by the `borderThickness` property.

To hide the bar, set `barTopMargin` to 1.0 and `barBottomMargin` to 1.0.

The following example shows how to customize the background and border colors provided by the default task item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myTaskItemStyle {
      backgroundColor : #00FF00;
      rollOverColor : #77FF77;
      selectedColor : #00FF00;
      selectedRollOverColor : #77FF77;
      borderThickness : 2;
      borderColor : #7A8E75;
      borderRollOverColor : #7A8E75;
      borderSelectedColor : #FFA53F;
      borderSelectedRollOverColor : #FFA53F;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}"
                      taskItemStyleName="myTaskItemStyle"/>
</mx:Application>
```

This example specifies the name of the CSS classes to be used to style the task item renderers for leaf and summary tasks. The styles for these CSS classes reproduce the default layout of leaf and summary tasks and customize the width of the border and the colors of the bars for leaf and summary tasks.

Start and end symbol properties

The start and end symbols mark the start and end time of a task. The start symbol is displayed centered at the start time of a task. Both its width and its height are equal to the height of the task item renderer. The end symbol is displayed centered at the end time of a task.

The default skin for symbols provides different shapes. Use the `startSymbolShape` property and `endSymbolShape` property to select the shape of the start and end symbols respectively.

Shape	Name
- (hidden)	`none`
⬟	`upPentagon`
⯂	`downPentagon`
◆	`diamond`
▲	`upTriangle`
▼	`downTriangle`
⇧	`upArrow`
⇩	`downArrow`

To hide the start symbol or end symbol, set the `startSymbolShape` or `endSymbolShape` to `none`.

The rendering of the start symbol can be customized with the following properties: `startSymbolBorderColor`, `startSymbolBorderThickness`, `startSymbolColor`, `startSymbolShape`, and `startSymbolSkin`. For the end symbol, the corresponding properties are: `endSymbolBorderColor`, `endSymbolBorderThickness`, `endSymbolColor`, `endSymbolShape`, and `endSymbolSkin`.

The following sample shows how to customize the start and end symbols of the default task item renderer for summary tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myTaskItemStyle {
      startSymbolColor: #FF0000;
      startSymbolBorderColor: #880000;
      startSymbolShape: up-arrow;
      endSymbolColor: #00FF00;
      endSymbolBorderColor: #008800;
      endSymbolShape: down-arrow;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}"
                      taskItemStyleName="myTaskItemStyle"/>
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the task item renderers for summary tasks and milestones. The styles for these CSS classes reproduce the default layout of summary tasks and milestones and customize the shape and colors of the start and end symbols of summary tasks as well as the start symbol for milestones.

Task text properties

The text is a label that provides a written description for a task. You can customize the styling of the text with the `textStyleName` and `textPosition` properties.

The following example shows how to customize the text style provided by the default task item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myTaskItemStyle {
      textStyleName: "myTextStyle";
      textRollOverColor: white;
      textSelectedColor: green;
      textSelectedRollOverColor: blue;
    }
    .myTextStyle {
      color: red;
      fontWeight: bold;
    }

  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}"
                      taskItemStyleName="myTaskItemStyle"/>
</mx:Application>
```

This example specifies the name of the CSS class to be used to style the task item renderer. The style for this CSS class adds start and end symbols to tasks.

Roll over and selection feedback

The default task item renderer allows you to customize the colors of the task bar, symbols and text for the interaction states: selected, rolled over, selected and rolled over.

The following table shows the style properties used for coloring the elements of the task item renderer depending on the selection state.

Element	Default	Selected	Rolled over	Selected and rolled over
bar background	`backgroundColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
start symbol background	`startSymbolColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
end symbol background	`endSymbolColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
bar border	`borderColor`	`borderSelectedColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
start symbol border	`startSymbolBorderColor`	`startSymbolBorderColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
end symbol border	`endSymbolBorderColor`	`endSymbolBorderColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
text	`color`	`textSelectedColor`	`textRollOverColor`	`textSelectedRollOverColor`

Animating a resource chart

Describes how to use the built-in animations in a resource chart.

In this section

Time navigation
Describes how to use the built-in animation when navigating in time.

Expand and Collapse
Describes how to use the built-in animations when expanding and collapsing tasks.

Time navigation

The Gantt sheet and the time scale provide built-in support for animating operations related to time navigation. For example, when users zoom in or out, pan to a given time, or focus on a given time range, all of these operations can be animated.

Note: By default, all of the built-in interactions provided by the Gantt sheet and time scale are animated.

The animation settings for time navigation are held by the Gantt sheet. They control both the animation of time navigation in the Gantt sheet and animation in the time scale.

For more details of the available animation-related styles, see *Animation properties*.

The `GanttSheet` component also allows you to animate the following time navigation operations through its API by setting the `animate` parameter.

```
moveTo(time:Date, animate:Boolean = false):void
showAll(margin:Number = 10, animate:Boolean = false):void
zoom(ratio:Number, time:Date = null, animate:Boolean = false):void
```

Expand and Collapse

The Gantt sheet leverages the built-in support of the Gantt data grid for animating the expanding and collapsing of items, and provides a similar animation for its tasks. It synchronizes the position and visibility of tasks with the position and visibility of the corresponding resources when they are expanded or collapsed in the Gantt data grid either by user interaction or through the API.

For information on the built-in animation support leveraged by the Gantt data grid from the `AdvancedDataGrid` class, see the AdvancedDataGrid API.

Managing resource chart events

Describes how to manage events that are a result of user interactions with a resource chart.

In this section

Data grid events
Describes how to manage events that are a result of user interactions with the data grid in a resource chart.

Gantt sheet data events
Describes how to manage events that are a result of user interactions with the data on a Gantt sheet in a resource chart.

Gantt sheet navigation events
Describes how to manage events that are a result of users navigating in the Gantt sheet in a resource chart.

Data grid events

The Gantt data grid leverages all the event types provided by the `AdvancedDataGrid`. The `AdvancedDataGrid` generates several event types that let you respond to user interaction on its displayed rows of resources.

See *Handling events in a DataGrid control* in *Adobe Flex 3 Developer's Guide* and *AdvancedDataGridEvent* in the *Adobe Flex 3 Language Reference* for more information on the available events.

Handling resource selection

If you need to process events triggered by the selection of resources of the Gantt data grid, you can use the `AdvancedDataGrid` API. The following example shows how to process events in this way.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import mx.controls.Alert;
      import mx.events.ListEvent;

      [Bindable]
      public var resourcesCollection:ArrayCollection =
        new ArrayCollection([
          { id: "T1", name: "Geneva Team", type: "team", children: [
            { id: "R1", name: "Project Manager",
              location: "Geneva", type: "member" },
            { id: "R2", name: "Developer",
              location: "Geneva", type: "member" }]
          }
        ]);

      [Bindable]
      public var resources:HierarchicalData =
        new HierarchicalData(resourcesCollection);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection ([
        { id: "T1", resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { id: "T2", resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { id: "T3", resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" },
        { id: "T4", resourceId: "R2", name: "Implement project",
          startTime: "1/21/2008 17:0:0", endTime: "1/30/2008 17:0:0" }
      ]);
      private function handleDataGridChange(event:ListEvent):void {
        Alert.show(event.itemRenderer.data['name'] + " was selected!",
                   "Change Event");
      }
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid change="handleDataGridChange(event)">
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:ResourceChart>
</mx:Application>
```

This example uses `ListEvent.CHANGE` triggered by the Gantt data grid to notify when a resource is selected. This basic example shows a message to the user with the name of the resource that was selected.

Expanding resource rows

If you need to expand or collapse resources of the Gantt data grid programmatically, you can use the `AdvancedDataGrid` API. The following example shows how to expand or collapse resources programmatically.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var resourcesCollection:ArrayCollection =
        new ArrayCollection([
          { id: "T1", name: "Geneva Team", type: "team", children: [
            { id: "R1", name: "Project Manager",
              location: "Geneva", type: "member" },
            { id: "R2", name: "Developer",
              location: "Geneva", type: "member" }]
          }
        ]);

      [Bindable]
      public var resources:HierarchicalData =
        new HierarchicalData(resourcesCollection);
      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection ([
        { id: "T1", resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { id: "T2", resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { id: "T3", resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" },
        { id: "T4", resourceId: "R2", name: "Implement project",
          startTime: "1/21/2008 17:0:0", endTime: "1/30/2008 17:0:0" }
      ]);


      public var genevaTeam:Object = resourcesCollection.source[0];
      public var expand:Boolean = true;
      private function expandCollapseResource():void
      {
        resourceChart.dataGrid.expandItem(genevaTeam, expand, true);
        expand = !expand;
      }
    ]]>
  </mx:Script>

  <mx:Button label="Expand Collapse Geneva Team"
             click="expandCollapseResource()"/>
  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:ResourceChart>
</mx:Application>
```

Gantt sheet data events

A GanttSheet component fires several types of events in response to user interactions with the data. You can create an event handler for each of these events.

You can listen for the following GanttSheet data events.

Gantt sheet data events

GanttSheet data event	Description
change	Indicates that the selection changed as a result of user interaction.
itemClick	Indicates that the user clicked the pointer over a visual item in the control.
itemDoubleClick	Indicates that the user double-clicked the pointer over a visual item in the control.
itemEditBegin	Indicates that the user started editing an item, either by dragging one of its extremities or by dragging it.
itemEditEnd	Indicates that the editing of an item has completed.
itemEditMove	Indicates that the user is moving an item.
itemEditReassign	Indicates that the user is reassigning a task to a resource.
itemEditResize	Indicates that the user is resizing an item.
itemRollOut	Indicates that the user rolled the pointer out of a visual item in the control.
itemRollOver	Indicates that the user rolled the pointer over a visual item in the control.

The Gantt sheet data events are of type GanttSheetEvent.

Within an event handler for a Gantt sheet data event you can access the data provider item associated with the event. You can also access the corresponding task item renderer and its TaskItem object as well as the GanttSheet object.

For information on using the task item editing events see *Editing tasks in a resource chart*.

In the following example a popup is displayed whenever a task is clicked, retrieving the label of the task from the TaskItem object.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import mx.collections.ArrayCollection;
      import mx.controls.Alert;
      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);

      public function onItemClick(event:GanttSheetEvent):void {
        var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
        Alert.show("You have clicked on the task:\n" + taskItem.label,
                   "Click Event");
      }
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet itemClick="onItemClick(event)" />
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

The previous example uses the `TaskItem` object to retrieve the label of the task. You can achieve the same result by retrieving the label directly from the `name` property of the data provider item.

```
     public function onItemClick(event:GanttSheetEvent):void {
        Alert.show("You have clicked on the task:\n" + event.item.name,
                   "Click Event");
      }
```

For more information on the `TaskItem` class see *Task item renderer architecture in resource chart*.

Gantt sheet navigation events

The Gantt sheet control fires an event of type `visibleTimeRangeChange` when the visible time range is modified, typically as a result of a user interaction.

Gantt sheet navigation event

GanttSheet navigation event	Description
visibleTimeRangeChange	Indicates that the visible time range has changed

The Gantt sheet visible time range event is of type `GanttSheetEvent`.

During animation of the change in the visible time range, a listener receives a `visibleTimeRangeChange` event for each animation step. You can use the `adjusting` property of the `GanttSheetEvent` object to determine whether the event is occurring during an animation. Each intermediate event has the `adjusting` property set to `true`; this property is set to `false` for the last event of an animation or when the change is not animated.

The following example shows how to display the visible time range whenever it is modified.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.gantt.GanttSheetEvent;
      import mx.controls.Alert;

      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);

      public function onVisibleTimeRangeChange(
                                  event:GanttSheetEvent):void {

        // Ignore the intermediate events generated during animation or
        // during user interactions.
        if (event.adjusting)
          return;
        Alert.show("The visible time range is now:\n" +
           "from " + GanttSheet(event.target).visibleTimeRangeStart+
           "\nto " + GanttSheet(event.target).visibleTimeRangeEnd,
           "VisibleTimeRangeChange Event");
      }
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet
        visibleTimeRangeChange="onVisibleTimeRangeChange(event)"/>
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example, if the `visibleTimeRangeChange` event is dispatched during animation the event is ignored.

User interaction with resource charts

Describes possible user interactions with resource charts.

In this section

User interaction in the Gantt data grid
Describes possible user interactions with the data grid in a resource chart.

User interaction in the time scale
Describes the facilities for navigation in time and control of the time range.

User interaction in the Gantt sheet
Describes the facilities for navigation in the Gantt sheet and for interacting with tasks.

User interaction in the Gantt data grid

The user interaction support available in the Gantt data grid, such as selection, sorting, editing, and the drag-and-drop feature, is provided entirely by the built-in behavior of `AdvancedDataGrid`.

See *DataGrid control user interaction* and *Sorting data in DataGrid controls* in *Adobe Flex 3 Developer's Guide* for more information on the possible built-in user interactions of `AdvancedDataGrid`.

Resource editing

The following example shows how to enable the editing of resources in the Gantt data grid by using its `editable` property.

This example sets the `editable` property of the Gantt data grid to `true` to enable its editing features. The example also displays two columns `Name` and `Location`. The `Name` column is editable and the `Location` column is read-only. These characteristics are achieved by setting the `editable` property of each column appropriately.

User interaction in the time scale

The time scale allows users to navigate in time and to control the time range displayed by the resource chart by using standard user interactions such as mouse clicks and dragging objects with the mouse.

The `GanttSheet` control broadcasts `visibleTimeRangeChange` events whenever the visible time range is modified.

Mouse navigation

The time scale has the mouse navigation features shown in the following table.

Mouse Navigation features in the time scale

Action	Description of action	Mouse or key and mouse action
Highlight	Highlights the hovered time interval.	Move.
Focus	Focuses on the selected time interval and makes it entirely visible.	Click.
Scroll	Scrolls the visible time range forward or backward in time.	Drag.
Focus interval	Focuses on an arbitrary time interval and makes it entirely visible.	Hold down the CTRL key and drag.
Zoom	Zooms the visible time in or out. The time under the mouse pointer is the same before and after zooming.	Hold down the CTRL key and rotate the mouse wheel forward or backward.

Note:
1. Navigation is limited by the `minVisibleTime` or `maxVisibleTime` value.
2. Zooming is limited by the `minZoomFactor` or `maxZoomFactor` value and the `minVisibleTime` or `maxVisibleTime` value, depending on which is the most constraining.

Attempting to zoom or navigate beyond these limits has no effect on the resource chart.

The time scale has the keyboard navigation features shown in the following table.

Keyboard navigation features in the time scale

Interaction	Description
+	Zooms the visible time in.
-	Zooms the visible time out.
Keyboard arrows	Lets the user move up, down, or from side to side in the Gantt sheet

User interaction in the Gantt sheet

The Gantt sheet and tasks respond to mouse gestures to provide end users with interactions.

Interactions with the Gantt sheet

The following table shows the mouse interactions available in the Gantt sheet.

Mouse interactions in the Gantt sheet

Interaction	Action
Hold down the CTRL key and rotate the mouse wheel forward or backward	Zooms the visible time in or out. The time under the mouse pointer is the same before and after zooming.
Drag	Lets the user move up, down, or from side to side in the Gantt sheet.

The `GanttSheet` control broadcasts `visibleTimeRangeChange` events when the visible time range is modified by user interaction.

The Gantt sheet has the mouse navigation features shown in the following table.

Keyboard navigation features in the Gantt sheet

Interaction	Description
+	Zooms the visible time in.
-	Zooms the visible time out.
Keyboard arrows	Lets the user move up, down, or from side to side in the Gantt sheet

Interactions with tasks

The `GanttSheet` control broadcasts:

- `change` events when the selection is modified by a user interaction.
- `itemEditBegin`, `itemEditMove`, `itemEditReassign`, `itemEditResize`, `itemEditEnd` events when the user moves, reassigns or resizes tasks.

The following table shows the mouse interactions available on tasks.

Mouse interactions on tasks

Interaction	Action
Hover	Highlights the task.
Click	Selects the task.
Drag the task horizontally	Moves the tasks along the time axis. The start and end time are adjusted.
Drag the task vertically	Reassigns the task to another resource.
Drag the start or end of the task bar	Changes the start or end time of the task.
Hold down the CTRL key and click	Extends or reduces the selection by toggling the selection state of the selected task. You must set the `allowMultipleSelection` property to `true` to allow multiple selection.

Editing tasks in a resource chart

Describes the task editing process and how to use events associated with task editing when the user moves, resizes or reassigns tasks in the resource chart control by direct manipulation.

In this section

About task editing in a resource chart
Discusses task editing in a resource chart in general.

Task editing process
Gives the sequence of steps when a task is edited.

Using task editing events
Describes the editing events and how to use them to customize editing.

Updating the data provider item
Describes the use of the `commitItem()` method to update the data provider item.

Accessing task data and the task item renderer in an event listener
Describes how a Gantt sheet event listener can access task data and the task item renderer.

Determining the kind of edit in an event listener
Describes how an event listener can determine the kind of editing action in progress.

Determining the reason for an itemEditEnd event
Describes how to determine the reason for an event signaling the end of an edit session.

Examples of editing event handlers
Gives several examples demonstrating the use of editing events.

About task editing in a resource chart

Editing tasks in a Gantt sheet has some similarities with editing cells in Flex® list-based components. However there are also differences: there is no item editor when you edit tasks and there is no `itemEditBeginning` event; you receive intermediate events for each change while editing a task.

The `moveEnabled`, `reassignEnabled` and `resizeEnabled` properties of the Gantt sheet in a resource chart control whether the user can move, reassign and resize the tasks. The `moveEnabledFunction`, `reassignEnabledFunction` and `resizeEnabledFunction` properties specify a custom function (if any) to control the editing capabilities for a specific task.

To represent the task being edited during editing, the Gantt sheet uses either the default `TaskItemRenderer` object or the task item renderer specified by the `taskItemRenderer` property.

For more information on the task item renderer see *Task item renderer architecture in resource chart*.

Task editing process

The following sequence of steps occurs when a task is edited in the Gantt sheet:

1. The user holds down the mouse button on a task and starts dragging.

   The kind of editing operation performed depends on which area of the task is being dragged:

   - ♦ The start extremity – when dragged, resizes the task by modifying its start time.
   - ♦ The end extremity – when dragged, resizes the task by modifying its end time.
   - ♦ The body of the task – when dragged, moves the task along the time axis or reassigns the task to another resource.

2. The Gantt sheet dispatches the `itemEditBegin` event and shows the editing tooltip. You can use this event to prepare data for editing.

   The next two steps can be repeated any number of times while the user is dragging.

3. The user moves or resizes the task.

4. The Gantt sheet dispatches the `itemEditMove`, `itemEditReassign` or `itemEditResize` event. You can use these events to modify the task that is being edited. For more information, see *Enforcing a minimum duration when resizing tasks*.

5. The user ends the editing session. Typically the task editing session ends when the user releases the mouse button.

6. The Gantt sheet dispatches the `itemEditEnd` event to update the data item and the Gantt sheet. You can use this event to modify the task item before it is saved.

7. The task that was being edited is updated in the Gantt sheet.

Using task editing events

A Gantt sheet component dispatches the following events as part of the task editing process: the `itemEditBegin`, `itemEditMove`, `itemEditReassign`, `itemEditResize` and `itemEditEnd` events. The Gantt sheet control defines default event listeners for all of these events except `itemEditBegin`.

You can write your own event listeners for one or more of these events to customize the editing process. When you write your own event listener, it executes before the default event listener defined by the component; the default listener executes afterward.

You can also replace the default event listener for the component with your own event listener. To prevent the default event listener from executing, you call the `preventDefault()` method from anywhere in your event listener.

Use the following events when you want to customize the editing of tasks:

itemEditBegin

Dispatched when the editing session starts. The user has pressed the mouse button over a task and started dragging.

The `GanttSheet.editKind` property is set with the editing operation being performed:

- ♦ `TaskItemEditKind.MOVE_REASSIGN` – the user is dragging the task body
- ♦ `TaskItemEditKind.RESIZE_START` – the user is dragging the start extremity of the task
- ♦ `TaskItemEditKind.RESIZE_END` – the user is dragging the end extremity of the task

The Gantt sheet component does not have a default listener for the `itemEditBegin` event. After the event is sent the Gantt sheet displays the editing tool tip which will provide information on the task while it is being edited. For more information see *Defining the tooltips for tasks*.

You can write an event listener for this event to modify the `TaskItem` data used by the task item renderer, or to modify some other information that will be used during editing. For more information see *Accessing task data and the task item renderer in an event listener*.

itemEditMove

Dispatched when the task is being moved along the time axis. The Gantt sheet component has a default listener for the `itemEditMove` event that adjusts the start and end time of the `TaskItem`.

The default event listener performs the following actions:

- ♦ Rounds the start time to the precision defined by the `snappingTimePrecision` property of the `GanttSheet` object.
- ♦ Updates the end time so the duration of the task remains constant.

You can write an event listener for this event to examine and modify the `TaskItem` data used by the task item renderer. For example, you can adjust the start and end time to take into account nonworking periods and maintain a constant work duration for the task. For more information see *Accessing task data and the task item renderer in an event listener*.

You can also implement a custom snapping policy or implement specific constraints on the start and end time. In this case you need to call `preventDefault()` to stop the Gantt sheet component from applying its own snapping policy or overriding your adjustments to the start and end time. For more information see *Custom snapping when moving tasks*.

itemEditReassign

Dispatched when the task is being reassigned to a different resource. The Gantt sheet component has a default listener for the `itemEditReassign` event that changes the resource of the `TaskItem` object. The default event listener performs the following action:

Calls the `acceptReassign` method of the `GanttSheet` to always accept the reassignment of the task to the hovered resource.

You can write an event listener for this event to accept or refuse the reassignment depending on conditions of your choice. For example, you can have resources of different types that can handle only tasks of a certain type or other similar constraints. In this case you need to call `preventDefault()` to stop the Gantt sheet from unconditionally accepting the reassignment, and call `acceptReassign` only when your criteria are met.

itemEditResize

Dispatched when the task is being resized.

The Gantt sheet component has a default listener for the `itemEditResize` event that adjusts the start or end time of the `TaskItem` object.

The default event listener performs the following actions:

♦ Uses the `snappingTimePrecision` property of the Gantt sheet to round the start or end time, depending on the kind of edit operation.

♦ Makes sure that the start time is before the end time.

You can write an event listener for this event to modify the `TaskItem` data used by the task item renderer. For example, you can adjust the start or end time to enforce a minimum duration; or you can implement a custom snapping policy; or you can implement other specific constraints on the time values. In these cases you need to call `preventDefault()` to stop the Gantt sheet from applying its own snapping policy or overriding your adjustments to the start and end time. For more information, see *Enforcing a minimum duration when resizing tasks*.

itemEditEnd

Dispatched when the task editing session ends, typically when the user releases the mouse button.

The Gantt sheet component has a default listener for this event that updates the item in the task data provider with the values from the `TaskItem` object used by the task item renderer.

The default event listener performs the following actions:

♦ When the edit operation was not cancelled the default event listener calls the `commitItem()` method to update the data provider item from the task item..

After all listeners of `itemEditEnd` have been called, the Gantt sheet component forces an update of the task by calling `ICollectionView.itemUpdated()` for this task data provider item, refreshing the Gantt sheet.

Updating the data provider item

To save changes made in the Gantt sheet back to the data provider, the default event listener calls the `commitItem()` method. This method uses the `ResourceChart.taskStartTimeField` and `ResourceChart.taskEndTimeField` properties to determine which properties of the data provider item must be updated and stores the edited start and end time in those fields. This behavior applies even if the `ResourceChart.taskStartTimeFunction` or `ResourceChart.taskEndTimeFunction` properties are defined.

When the data provider item is an XML object, the values of `startTime` and `endTime` are stored as Strings using the `Date.toString` method. When the data provider item is an ActionScript® object, the values of the `startTime` and `endTime` are stored as `Date` objects.

You can override this behavior by setting the `commitItemFunction` property to point to a custom `commitItem()` function.

You typically customize `commitItem()` to perform the following actions:

♦ Store the start and end time in a custom type inside the task data provider. Whenever you store the start and end time with custom types, you must make sure that the task chart handles the types you use or provide your own functions specified with the `taskStartTimeFunction` and `taskEndTimeFunction` properties.

♦ Store custom properties, other than the start time and end time that have been modified during editing.

For more information on using the `commitItemFunction` see *Updating the data provider item using a custom date format*.

Accessing task data and the task item renderer in an event listener

From within a Gantt sheet event listener you have access to the task data provider item, the current `startTime` and `endTime` of the task, the associated resource data provider item, and the item renderer used to display the task.

To access the data provider item you use the `item` property of the event.

To access the item renderer you use the `itemRenderer` property of the event.

To access the current `startTime` and `endTime`, or the resource associated with the task, you use the `itemRenderer.data` property of the event.

For more details on the task item renderer and how it accesses data see *Task item renderer architecture in resource chart*.

The following example shows an event listener for the `itemEditMove` event that accesses the data provider item, the task item renderer and the modified start and end time of the task.

```
  // The renderer
   var renderer:IDataRenderer = event.itemRenderer;
private function onItemEditMove(event:GanttSheetEvent):void
{
   //...


   // The TaskItem that holds the modified values of the task
   var taskItem:TaskItem = event.itemRenderer.data as TaskItem;

   // Modified values of the task
   var startTime:Date = taskItem.startTime;
   var endTime:Date = taskItem.endTime;
   var resourceId:String = taskItem.resourceId;

   // The data provider item
   var item:Object = event.item;

   // ...
}
```

Determining the kind of edit in an event listener

When the user starts editing a task, the Gantt sheet component determines the kind of edit operation from the item area where the user clicked. In the body of the event listeners for the `itemEditBegin`, `itemEditMove`, `itemEditReassign`, `itemEditResize`, and `itemEditEnd` events, you can determine the kind of edit and then handle it accordingly.

The `GanttSheet` class defines the `editKind` property, which contains a value that indicates the type of edit operation in progress. The `editKind` property has the following values defined in the class `TaskItemEditKind`.

Values of the kind of edit

Value	Description
MOVE_REASSIGN	Specifies that the user has clicked in the body of the task and is moving or reassigning the task. You may get this value for `itemEditBegin`, `itemEditMove`, `itemEditReassign` and `itemEditEnd` events.
RESIZE_END	Specifies that the user has clicked the end extremity of the task and is resizing it by changing the end time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.
RESIZE_START	Specifies that the user has clicked the start extremity of the task and is resizing it by changing the start time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.

The following example shows how to retrieve the kind of edit within an event listener.

```
private function onItemEditBegin(event:GanttSheetEvent):void {
      var editKind:String = GanttSheet(event.target).editKind;
      //...
    }
```

Determining the reason for an itemEditEnd event

A user can end a task editing operation in several ways. In the body of the event listener for the `itemEditEnd` event, you can determine the reason for the event, and then handle it accordingly.

The `GanttSheetEvent` class defines the `reason` property, which contains a value that indicates the reason for the event. The `reason` property has the following values defined in the class `GanttSheetEventReason`.

Values of the reason property

Value	Description
CANCELLED	Specifies that the user canceled editing and that they do not want to save the edited data.
COMPLETED	Specifies that the user completed the edit session, usually by releasing the mouse button.
OTHER	Specifies that the Gantt sheet is somehow in a state where editing is not allowed.

Examples of editing event handlers

Enforcing a minimum duration when resizing tasks

The following example in MXML shows how to use the `itemEditResize` event to enforce a minimum duration when resizing tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import ilog.gantt.TaskItemEditKind;
      import ilog.gantt.TimeUtil;
      import ilog.utils.GregorianCalendar;
      import ilog.utils.TimeUnit;

      import mx.collections.ArrayCollection;
      import mx.events.FlexEvent;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);

      private function onItemEditResize(event:GanttSheetEvent):void
      {
        // Prevent the default behavior.
        event.preventDefault();

        var ganttSheet:GanttSheet = event.target as GanttSheet;
        var calendar:GregorianCalendar = ganttSheet.calendar;

        var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
        var startTime:Date = taskItem.startTime;
        var endTime:Date = taskItem.endTime;
        var duration:Number;

        if (ganttSheet.editKind == TaskItemEditKind.RESIZE_START) {
          // Snaps the edited start time on the
          // snapping time precision.
          startTime = calendar.round(startTime,
                                     ganttSheet.snappingTimePrecision.unit,
                                     ganttSheet.snappingTimePrecision.steps);


          // Ensure that the duration is at least 1 day.
          duration = endTime.time - startTime.time;
          if (duration < TimeUnit.DAY.milliseconds) {
            startTime.time = endTime.time - TimeUnit.DAY.milliseconds;
          }
          taskItem.startTime = startTime;
        }

        if (ganttSheet.editKind == TaskItemEditKind.RESIZE_END) {
          // Snaps the edited end time on the
          // snapping time precision.
          endTime = calendar.round(endTime,
                                   ganttSheet.snappingTimePrecision.unit,
                                   ganttSheet.snappingTimePrecision.steps);

          // Ensure that the duration is at least 1 day.
          duration = endTime.time - startTime.time;
          if (duration < TimeUnit.DAY.milliseconds) {
            endTime.time = startTime.time + TimeUnit.DAY.milliseconds;
          }
          taskItem.endTime = endTime;
        }
      }
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet itemEditResize="onItemEditResize(event)" />
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example the listener for the itemEditResize event enforces the minimum duration of 1 day for tasks. It also mimics the behavior of the default listener by snapping the edited start time and end time using the snappingTimePrecision property of the GanttSheet object.

Custom snapping when moving tasks

The following example shows how to use the itemEditMove event to snap the start of a task when moving it.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
```

```
<mx:Script>
  <![CDATA[
    import ilog.gantt.GanttSheet;
    import ilog.gantt.GanttSheetEvent;
    import ilog.gantt.TaskItem;
    import ilog.utils.TimeUnit;
    import mx.collections.ArrayCollection;

    [Bindable]
    public var resources:ArrayCollection = new ArrayCollection([
      { id: "R1", name: "Project Manager", location: "Geneva" }
    ]);

    [Bindable]
    public var tasks:ArrayCollection = new ArrayCollection([
      { resourceId: "R1", name: "Define project vision",
        startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
      { resourceId: "R1", name: "Refine project vision",
        startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
      { resourceId: "R1", name: "Assess project vision",
        startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
    ]);

    private function onItemEditMove(event:GanttSheetEvent):void
    {
      // Prevent the default behavior.
      event.preventDefault();

      // Snaps the start time on days boundary, starting at 08:00
      var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
      var ganttSheet:GanttSheet = event.target as GanttSheet;
      var startTime:Date = ganttSheet.calendar.round(
                                          taskItem.startTime,
                                          TimeUnit.DAY, 1);
      startTime.hours = 8;

      // Move the item while preserving a constant duration.
      if (startTime.time != taskItem.startTime.time)
      {
        var duration:Number = taskItem.endTime.time
                            - taskItem.startTime.time;
        taskItem.startTime.time = startTime.time;
        taskItem.endTime.time = startTime.time + duration;
      }
    }
  ]]>
</mx:Script>

<ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                    resourceDataProvider="{resources}"
                    taskDataProvider="{tasks}">
  <ilog:ganttSheet>
    <ilog:GanttSheet itemEditMove="onItemEditMove(event)" />
  </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example the listener for the `itemEditMove` event snaps the start time of tasks to 08:00 for each day.

Updating the data provider item using a custom date format

The following example shows how to access dates stored with a custom date format. The start and end time are read from the data provider item using custom functions that parse the custom date format. A custom function is used to update the data provider item after editing is complete using the custom date format for the start time and end time.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import ilog.core.DataItem;
      import ilog.gantt.GanttSheetEventReason;
      import ilog.gantt.GanttSheet;
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import mx.formatters.DateFormatter;

      private var replaceDashes:RegExp = new RegExp("-","g");
      private var replaceT:RegExp = new RegExp("T");
      private var iso8601Formatter:DateFormatter = new DateFormatter();

      private function initApp():void {
        iso8601Formatter.formatString = "YYYY-MM-DDTJJ:NN:SS";
      }

      private function readStartDate(item:Object):Date {
        return readDate(item, "startTime");
      }

      private function readEndDate(item:Object):Date {
        return readDate(item, "endTime");
      }

      private function readDate(item:Object, dateField:String):Date {
        var dateString:String = item[dateField].replace(replaceDashes, "/");
        dateString = dateString.replace(replaceT, " ");
        return new Date(dateString);
      }

      // When setting taskStartTimeFunction or taskEndTimeFunction
      // you must provide a commitItemFunction to update the data
      // provider item. In this case we update the start and end
      // time in the data provider item using a ISO 8601 date
      // formatter.
      private function commitItem(dataItem:DataItem):void {
        if (dataItem is TaskItem)
        {
          var item:Object = dataItem.data;
          var taskItem:TaskItem = dataItem as TaskItem;

          item["startTime"] = iso8601Formatter.format(taskItem.startTime);
          item["endTime"] = iso8601Formatter.format(taskItem.endTime);
          item["resourceId"] = taskItem.resourceId;
        }
      }
    ]]>
  </mx:Script>

  <!-- The dates in the model are expressed using the ISO 8601 Extended
       Format for date and time of day representation. For more details
       on this format see:
       "http://isotc.iso.org/livelink/livelink/4021199/ISO_8601_2004_E.zip?
       func=doc.Fetch&nodeid=4021199"
   -->
  <mx:Model id="model">
    <model>
      <resource id="R1" name="Resource" />
      <task id="T1" resourceId="R1" name="Define project vision"
            startTime="2008-01-14T08:00:00" endTime="2008-01-15T17:00:00"/>
      <task id="T2" resourceId="R1" name="Refine project vision"
            startTime="2008-01-16T08:00:00" endTime="2008-01-17T17:00:00"/>
      <task id="T3" resourceId="R1" name="Assess project vision"
            startTime="2008-01-18T18:00:00" endTime="2008-01-21T17:00:00"/>
    </model>
  </mx:Model>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{model.resource}"
                      taskDataProvider="{model.task}"
                      taskStartTimeFunction="readStartDate"
                      taskEndTimeFunction="readEndDate">
    <ilog:ganttSheet>
      <ilog:GanttSheet commitItemFunction="commitItem" />
    </ilog:ganttSheet>
  </ilog:ResourceChart>
</mx:Application>
```

In this example the start and end time of tasks are stored in the task data provider using the ISO8601:2000 extended format. Custom functions are set on the `taskStartTimeFunction` and `taskEndTimeFunction` properties of the resource chart control. These functions parse the ISO8601:2000 formatted String and retrieve Date objects, stored in the `TaskItem` object associated with the task data provider item. The `commitItemFunction` property of the Gantt sheet points to a custom function. This function uses a custom date formatter to store the start and end time as Strings in the IS8601:2000 format; it also updates the `resourceId` for the task.

Working with Date objects

IBM ILOG Elixir provides a set of utility classes to help manipulate `Date` objects:

- ♦ `ilog.utils.TimeUnit`, defines time units
- ♦ `ilog.utils.GregorianCalendar`, provides operations for accurately manipulating Date objects

The resource chart control uses an instance of `GregorianCalendar` that you can retrieve from the `ResourceChart.calendar` property. It is important to use the same instance of `GregorianCalendar` that is used by the resource chart control to achieve consistent results.

The `GregorianCalendar` class provides the following methods:

- ♦ `addUnits`, allows the adding of time units to a `Date` object or the removing of time units from a `Date` object.
- ♦ `floor`, gets the floor of a `Date` object for a given time unit.
- ♦ `getWeek`, returns the number of the week for a `Date` object.
- ♦ `round`, rounds a `Date` object to the nearest time unit, for a given time unit.

For complete reference information, see the `TimeUnit` class and the `GregorianCalendar` class.

For more information on `Date`, see *Working with dates and times* in *Adobe Programming ActionScript 3.0*.

Task charts

Describes how to use IBM ILOG Elixir task charts with Adobe® Flex® 3.

In this section

Introduction to task charts
Describes task charts and gives an example.

Task chart architecture
Describes the architecture of the IBM ILOG Elixir task chart classes and their relationship to the Adobe® Flex® 3 types..

Creating a task chart control
Describes how to define a task chart control.

Configuring task chart data
Describes how to configure task chart data.

Configuring a task chart control
Describes how to configure a task chart control.

Task chart item renderers
Describes the default item renderers used to display the graphic representation of task and constraint data items in the Gantt sheet, explains how the item renderers get the information they need to display tasks and constraints, and describes the use of custom renderers in the Gantt sheet and in the data grid.

Styling a task chart
Describes how to style the data grid, Gantt sheet, task items, and time scale in a task chart.

Animating a task chart
Describes how to use the built-in animations in a task chart.

Managing task chart events
Describes how to manage events that are a result of user interactions with a task chart.

User interaction with task charts
Describes possible user interactions with task charts.

Editing tasks in a task chart
Describes the task editing process and how to use events associated with task editing when the user moves or resizes tasks in the task chart control by direct manipulation.

Working with Date objects
Describes how to use the utility classes for manipulating Date objects.

Introduction to task charts

A task chart is a Gantt chart used to display tasks and their dependencies. It is typically found in project planning applications.

The `TaskChart` control is a list of tasks that display their properties in columns and their duration over a time line. Tasks may be organized as a hierarchy, either to match a hierarchy of tasks or to arrange them into groups or categories.

The task chart control is a composite control composed of:

♦ A data grid that displays the hierarchy of tasks and their properties. It uses the Adobe® Flex® `AdvancedDataGrid` control.

  You are assumed to be familiar with using the `AdvancedDataGrid` control and working with data providers. For more information on the `AdvancedDataGrid` control, see *Using the AdvancedDataGrid control* in *Adobe Flex 3 Developer's Guide*.

♦ A Gantt sheet that displays the tasks arranged in time. The Gantt sheet header displays a time scale that supports navigatiing and zooming on the time axis.

The Gantt sheet displays from back to front:

♦ A row grid to help visualize the rows

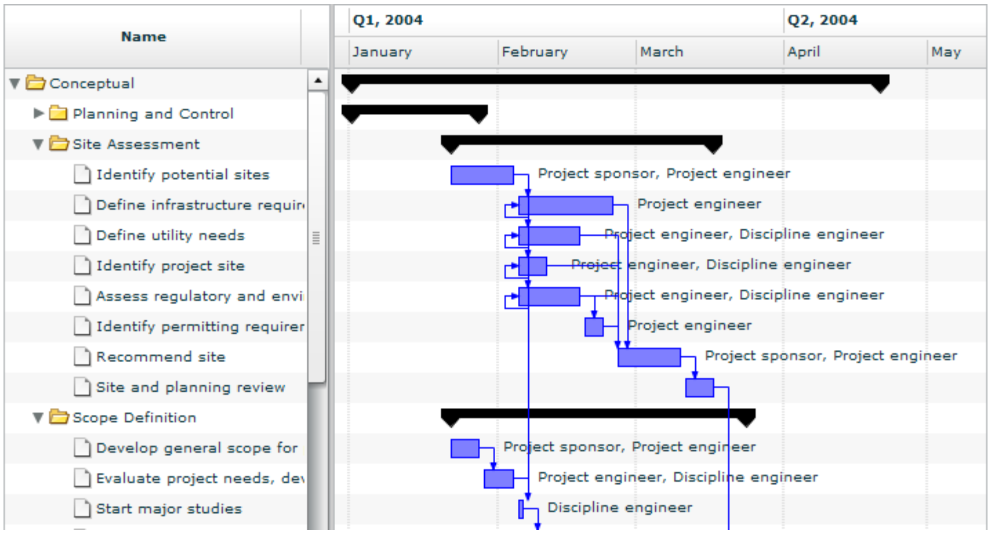
♦ A work grid that renders the working and nonworking times

♦ A time grid to help visualize the time units

♦ The tasks, and the dependencies or constraints between the tasks.

♦ A time indicator to show the current time

The following is an example of a task chart.


Name
Q1, 2004
Q2, 2004
January
February
March
April
May
Conceptual
Planning and Control
Site Assessment
Identify potential sites
Define infrastructure requir
Define utility needs
Identify project site
Assess regulatory and envi
Identify permitting requirer
Recommend site
Site and planning review
Scope Definition
Develop general scope for
Evaluate project needs, dev
Start major studies
Project sponsor, Project engineer
Project engineer
Project engineer, Discipline engineer
Project engineer, Discipline engineer
Project engineer, Discipline engineer
Project engineer
Project sponsor, Project engineer
Project sponsor, Project engineer
Project engineer, Discipline engineer
Discipline engineer

The task chart is characterized by the ability to:

- Display tasks as a hierarchy in a data grid by leveraging the features of the Adobe Flex `AdvancedDataGrid`.
- Display tasks over a time line inside a Gantt sheet.
- Edit task duration, start, and end times in response to direct manipulation with the mouse.
- Dispatch events related to hovering over, clicking, double-clicking, and editing tasks.
- Display constraints between tasks.
- Navigate along the time axis and focus on time periods.
- Skin and style tasks and constraints.

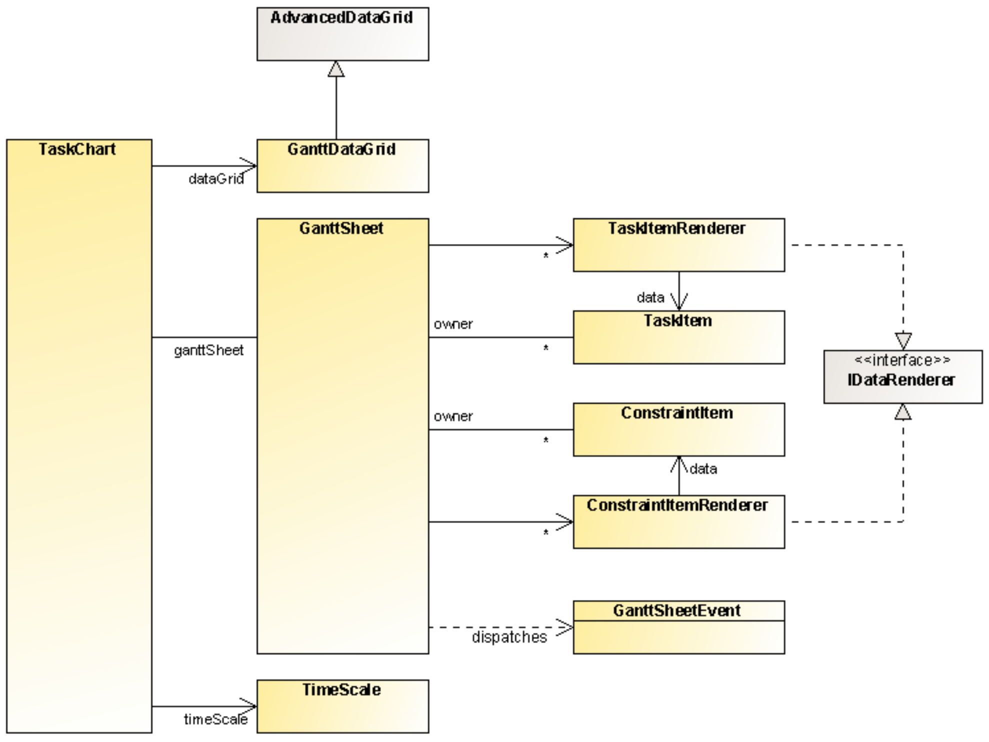
A *task* corresponds to some work to be completed. Tasks are often broken down into *subtasks*, thus forming a hierarchy. In this hierarchy, tasks that have no children are called *leaf task*s, while tasks that have children are called *summary tasks*.

A *milestone* is a special kind of task of zero duration that corresponds to an event.

A *constraint* is a condition set between two tasks. A constraint can be one of the following kinds: start to start, start to end, end to start, or end to end. In a constraint, the *source task* is the task whose start or end controls the start or end of another task. Conversely, the *destination task* is the task whose start or end depends on the start or end of another task. In the Gantt sheet, a constraint is represented by a polyline terminated by an arrow, linking the source task to the destination task.

Task chart architecture

The task chart architecture is shown in the following figure.



The task chart classes shown in the UML diagram are described in the following table.

Main task chart classes

Class	Description
`TaskChart`	The task chart component. It extends the `UIComponent` class. It is the top-level object to be used in MXML.
`GanttDataGrid`	The component that displays the data grid for the resources. The GanttDataGrid class extends the `AdvancedDataGrid` class.
`GanttSheet`	The component that displays the Gantt sheet. It extends the `UIComponent` class.
`GanttSheetEvent`	Defines the type of events dispatched by the Gantt sheet.
`TimeScale`	The component that displays the time scale. It extends the `UIComponent` class.
`TaskItemRenderer`	The default item renderer for tasks in the Gantt sheet.
`TaskItem`	The data that is set on the task item renderers.
`ConstraintItemRenderer`	The default item renderer for constraints in the Gantt sheet.
`ConstraintItem`	The data that is set on the constraint item renderer.

Creating a task chart control

Defining a task chart control in MXML

You can define a `TaskChart` control in MXML using the `<ilog:TaskChart>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value.

A `TaskChart` control displays task and constraint data and can be configured by using two data providers as shown in the following example.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" },
        { fromId:"T2", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T5", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  />
</mx:Application>
```

Configuring task chart data

A task chart control gets its data from two data providers: a task data provider containing the task data items, and a constraint data provider containing the constraint data items.

The task data provider can be a flat list or can contain hierarchical or grouped task data. It is set using the `taskDataProvider` property of the `TaskChart` object. The data grid of the task chart leverages all features of the `AdvancedDataGrid` it is built on, and most considerations regarding the definition of data for an `AdvancedDataGrid` object apply to the task data provider. For more information see *Hierarchical and grouped data display* in *Using the AdvancedDataGrid control* in *Adobe Flex 3 Developer's Guide*.

Note: You must set the task data provider using the `taskDataProvider` property on the `TaskChart` instance. Setting a data provider on the data grid will have no effect.

The constraint data provider is a flat list of constraints. It is set using the `constraintDataProvider` property of the `TaskChart` object. For more information see *The constraint data provider*.

In addition to specifying the task data provider and the constraint data provider of the task chart control, you must also:

♦ Specify the fields of the task items that are used as the columns in the data grid. For more details see *Specifying task columns*.

♦ Specify the fields of the task items that are used to represent the tasks in the Gantt sheet. For more details see *Specifying task data items with field mappings*.

♦ Specify the fields of the constraint items that are used to represent the constraint in the Gantt sheet. For more details see *Specifying constraint data items with field mappings*
.

The constraint data provider

The constraint data provider is a list-based data provider. For more information on list-based data providers see *Using Data Providers and Collections* in the *Adobe Flex 3 Developer's Guide*.

You specify the constraint data for the `TaskChart` control by using the `constraintDataProvider` property.

The constraint data provider supports the following types of data:

XML
A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag, or an `XML` or `XMLList` object.

IList or ICollectionView
An object that implements the `IList` or `ICollectionView` interface (such as an instance of the `ArrayCollection` or `XMLListCollection` class), whose data provider conforms to the structure specified in either of these items.

Other objects
An array of items or an object.

When setting the `constraintDataProvider` property the `TaskChart` control modifies its internal representation of the constraint data as follows:

♦ XML or XMLList are wrapped in an instance of `XMLListCollection`.

♦ `IList` or `ICollectionView` are used directly.

♦ An `Array` object is wrapped in an instance of `ArrayCollection`.

♦ An ActionScript® object of any other data type is wrapped in an `ArrayCollection` using an `Array` containing the object as its sole entry.

For example, you pass an `Array` to the `constraintDataProvider` property. If you read the data back from the `constraintDataProvider` property it is returned as an instance of `ArrayCollection`.

The best practice when your constraint data can change dynamically is to use a collection, which provides the necessary notifications of changes.

Specifying task columns

To specify the task properties that are shown on the Gantt data grid, map the properties of a task to columns. This mapping follows the scheme defined by `AdvancedDataGrid`.

The following example shows how to do this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" },
        { fromId:"T2", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T5", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
          <mx:AdvancedDataGridColumn dataField="id"
                                     headerText="ID"/>
          <mx:AdvancedDataGridColumn dataField="startTime"
                                     headerText="Start"/>
          <mx:AdvancedDataGridColumn dataField="endTime"
                                     headerText="End"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:TaskChart>
</mx:Application>
```

This example displays the Name, ID, Start, and End columns in the Gantt data grid, mapped from the `name`, `id`, `startTime` and `endTime` properties of task items.

See *Specifying columns* in *Creating a DataGrid control* in *Adobe Flex 3 Developer's Guide* for more information.

Specifying task data items with field mappings

In addition to specifying the `taskDataProvider` of the `TaskChart` control, to display (or render) the tasks you must map the fields of task data items (start time, end time, and label) to fields that exist in the `taskDataProvider` by setting field properties in the control. In the task data provider, a field is either a property (if the task data provider items are ActionScript objects) or an attribute (if the task data provider items are XML objects).

The following table shows the field properties in a `TaskChart` control that contain the field names used to map task data. In each case, the first default value of the field name is for an ActionScript object and the second (with `@` prefix) is for an XML object.

Field properties for the tasks in the TaskChart control

Property	Default value	Purpose
`taskEndTimeField`	`endTime` or `@endTime`	The value of the field whose name is stored in this property is used

Property	Default value	Purpose
		as the end time of each task data item.
`taskIsMilestoneField`	`milestone` or `@milestone`	The value of the field whose name is stored in this property is used to determine whether a task is a milestone.
`taskIsSummaryField`	`summary` or `@summary`	The value of the field whose name is stored in this property is used to determine whether a task is a summary task.
`taskLabelField`	`name` or `@name`	The value of the field whose name is stored in this property is used to display the label of each task data item.
`taskStartTimeField`	`startTime` or `@startTime`	The value of the field whose name is stored in this property is used as the start time of each task data item.

The value of fields `taskEndTimeField` and `taskStartTimeField` must be either a `Date` object or any `String` representation of a date that is supported by the constructor of `Date` objects.

The value of the fields `taskIsMilestoneField` and `taskIsSummaryField` must be either a Boolean object or a String with the possible values: `true` or `false`. By default tasks that have children in the task data provider are considered to be summary tasks.

Specifying task data items with custom functions

If you need an alternative way of retrieving the values of task data items, you can use the `taskEndTimeFunction`, `taskLabelFunction`, `taskStartTimeFunction`, `taskIsMilestoneFunction` and `taskIsSummaryFunction` properties of the control to specify custom functions that will be used to obtain the end time, label, start time, milestone flag, and summary flag respectively of each task data item from the `taskDataProvider`.

The following example shows how you can use your own date-parsing function to read the start and end time in XML.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
    <mx:Script>
      <![CDATA[
        import mx.collections.HierarchicalData;

        [Bindable]
        public var tasks:HierarchicalData = new HierarchicalData([
          { id: "T1", name: "Task #1",
            startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
          { id: "T2", name: "Milestone #1", milestone: "true",
            startTime: "1/29/2008 8:0:0", endTime: "1/29/2008 08:0:0"}
        ]);

        public function parseDate(value:String):Date {
          return new Date(value);
        }
        public function startTimeFunction(item:Object):Date {
          return parseDate(item.startTime);
        }
        public function endTimeFunction(item:Object):Date {
          return parseDate(item.endTime);
        }
        public function isMilestoneFunction(item:Object):Boolean {
          return new Boolean(item["milestone"]);
        }
      ]]>
    </mx:Script>

    <ilog:TaskChart width="100%" height="100%"
                        taskDataProvider="{tasks}"
                        taskStartTimeFunction="{startTimeFunction}"
                        taskEndTimeFunction="{endTimeFunction}"
                        taskIsMilestoneFunction="{isMilestoneFunction}">
    </ilog:TaskChart>
</mx:Application>
```

In this example the `parseDate` function mimics the default behavior when parsing dates. The `isMilestoneFunction` mimics the default behavior used to determine whether a task is a milestone. You can use the `taskLabelFunction` property to specify a function that returns a label depending on the item in a similar way. Likewise you can use the `taskIsSummaryFunction` property to specify a function that returns a Boolean indicating whether the task is a summary task.

Specifying constraint data items with field mappings

In addition to specifying the constraint data provider for the `TaskChart` control in the `constraintDataProvider` property, to display the constraints you must map the fields of the constraint data items to fields that exist in the constraint data provider by setting field properties in the control. In the constraint data provider, a field is either a property (if the constraint data provider items are ActionScript objects) or an attribute (if the constraint data provider items are XML objects).

The following table shows the field property in a `TaskChart` control that contains the field name used to map constraint data. The first default value of the field name is for an ActionScript object and the second one (with @ prefix) is for an XML object.

Property	Default value	Purpose
constraintKindField	`kind` or `@kind`	The value of the field whose name is stored in this property is used as the kind of each constraint data item.

The value of the field `constraintKindField` must be a String with one of the following possible values: `startToStart`, `startToEnd`, `endToStart`, `endToEnd`.

Specifying constraint data items with custom functions

If you need an alternative way of retrieving the values of constraint data items, you can use the `constraintKindFunction` property of the control to specify a custom function that will be used to obtain the kind of constraint for each constraint data item from the constraint data provider.

Specifying the relationship between tasks and constraints

The task chart control associates tasks and constraints as follows:

- One task can be the source or destination of zero or more constraints
- One constraint has one source task and one destination task

This association is built using fields of the data provider items which are specified with properties of the `TaskChart` class as shown in the following table.

Field properties in the task—constraint association

Property	Default value	Description
`taskIdField`	`id` or `@id`	A uniquer identifier of the task within the task data provider. This is a field of the task data provider items.
`constraintFromIdField`	`taskId` or `@taskId`	The identifier of the task that is the source of the constraint. This is a field of the constraint data provider items.
`constraintToIdField`	`taskId` or `@taskId`	The identifier of the task that is the destination of the constraint. This is a field of the constraint data provider items.

If changes are made to the relationship outside the task chart control, you must notify the task data provider for the tasks concerned by calling its `ICollectionView.itemUpdated()` method.

The `TaskChart` class provides the following methods to access the association:

- `getFromTask(constraint:Object):Object` — returns the task that is the source of the constraint.
- `getToTask(constraint:Object):Object` — returns the task that is the destination of the constraint.

- `getFromConstraints(task:Object):Array` — returns the constraints that have the task as their source.
- `getToConstraints(task:Object):Array` — returns the constraints that have the task as their destination.

For complete information on the properties and the associated methods see the `TaskChart` class.

Configuring a task chart control

Describes how to configure a task chart control.

In this section

Hiding task columns
Describes how to hide task columns in a task chart control.

Specifying task icons in the data grid
Describes how to change the task icons used in a task chart control.

Specifying the visible time range
Describes how to specify the visible time range in a task chart control.

Specifying the minimum and maximum visible time
Describes how to set the minimum and maximum visible time in a task chart control.

Specifying the minimum and maximum zoom factors
Describes how to set the minimum and maximum zoom factors in a task chart control.

Defining the tooltips for tasks and constraints
Describes the different tooltips for items (tasks or constraints) and how to define them.

Defining the working and nonworking periods
Describes how to customize the working and nonworking periods.

Snapping tasks when moving or resizing
Describes how to snap tasks when moving or resizing them.

Hiding task columns

To hide a column you can set its `visible` property to `false` as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
          <mx:AdvancedDataGridColumn dataField="startTime"
                                     headerText="Start"
                                     visible="false"/>
          <mx:AdvancedDataGridColumn dataField="endTime"
                                     headerText="End"
                                     visible="false"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:TaskChart>
</mx:Application>
```

In this example, you display only the `Name` column in the Gantt data grid, which is mapped from the `name` property of a task. The `Start` and `End` columns are no longer displayed.

See *Hiding and displaying columns* in *Creating a DataGrid control* in *Adobe Flex 3 Developer's Guide* for more information.

Specifying task icons in the data grid

You can change the icons used by the Gantt data grid in different ways, since it leverages the icon-customization support provided by the `AdvancedDataGrid` class.

The following example shows how to change the icons used for summary tasks and the icons used for leaf tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalCollectionView;
      import mx.collections.IHierarchicalCollectionView;
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import ilog.gantt.TaskItem;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Embed(source="summary.png")]
      public var SummaryIcon:Class;

      [Embed(source="leaf.png")]
      public var LeafIcon:Class;

      public function customIconFunction(item:Object):Class
      {
        var taskItem:TaskItem = taskChart.ganttSheet.itemToTaskItem(item);
        return taskItem.isSummary ? SummaryIcon : LeafIcon;
      }

    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
```

```
                  taskDataProvider="{tasks}"
                  >
    <ilog:dataGrid>
      <ilog:GanttDataGrid iconFunction="customIconFunction">
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:TaskChart>
</mx:Application>
```

This example shows how the `iconFunction` property of the `AdvancedDataGrid` object is used to set a custom function, `customIconFunction`, which checks the `isSummary` property of the processed item.

See *Hierarchical and grouped data display* for the `AdvancedDataGrid` control in *Adobe Flex 3 Developer's Guide* for more information on the various options for customizing the icons in an `AdvancedDataGrid` component.

Specifying the visible time range

The visible time range defines the range of time that is displayed in the Gantt sheet. You can specify the visible time range with the `visibleTimeRangeStart` and the `visibleTimeRangeEnd` properties of the Gantt sheet.

The following example shows how to set the visible time range from 1 January 2008 00:00:00 to 31 December 2008 24:00:00.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="600" height="400"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}">
    <ilog:ganttSheet>
      <ilog:GanttSheet
          visibleTimeRangeStart="{new Date(2008, 0, 1)}"
          visibleTimeRangeEnd="{new Date(2008, 11, 31, 24)}" />
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

When the visible time range is not set explicitly, the task chart displays the range of time corresponding to the data provided.

Specifying the minimum and maximum visible time

The minimum and maximum visible time define the navigable time range. The visible time range will always be within this navigable time range. Depending on your application, it can be convenient to limit the navigable time range to prevent end users from being lost in time. A typical read-only task chart should limit navigation to the range of available data.

The following example shows you how to set the minimum and maximum visible time from 1 January 2007 00:00:00 to 31 December 2010 24:00:00.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}">
    <ilog:ganttSheet>
      <ilog:GanttSheet minVisibleTime="{new Date(2007, 0, 1)}"
                       maxVisibleTime="{new Date(2010, 11, 31, 24)}"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

The minimum visible time should be on or after 1 January 100.

The maximum visible time should be on or before 1 January 10000.

Specifying the minimum and maximum zoom factors

The minimum and maximum zoom factors define the range of possible values of the zoom factor in the Gantt sheet. The zoom factor, expressed in milliseconds per pixel, defines the correspondence from time to screen space and conversely from screen space to time. It corresponds to the precision that users can obtain along the time axis.

For example, in an application where the precision of the start and end of tasks is shown in days, you can limit the space allocated to represent a day to a maximum of 30 pixels. The following example shows how to limit the space in this way.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.utils.TimeUnit;
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}">
    <ilog:ganttSheet>
      <ilog:GanttSheet
        minZoomFactor="{TimeUnit.DAY.milliseconds / 30}"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example, the maximum zoom factor is not explicitly set; it is left unbounded.

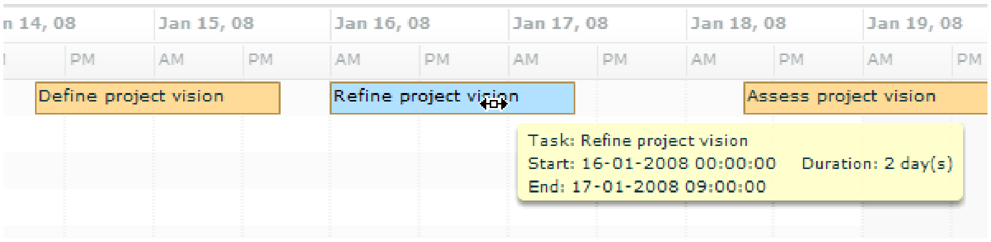
The minimum zoom factor should be greater than `TimeUnit.MILLISECONDS.milliseconds / 1`.

The maximum zoom factor should be less than `TimeUnit.DECADE.milliseconds / 20`.

Defining the tooltips for tasks and constraints

The Gantt sheet component uses two kinds of tooltip for items: a data tip displayed when the mouse pointer hovers over an item and an editing tip displayed while items are being edited to provide feedback on the ongoing operation.

The following image shows the default tooltip for a task.



Task: Refine project vision
Start: 16-01-2008 00:00:00 Duration: 2 day(s)
End: 17-01-2008 09:00:00

By default the data tip and the editing tip show the same information; for a task: the name of the task, its start and end time and its duration; for a constraint: the kind of constraint, the name of the source task and the name of the destination task. You can customize the contents of the data tip by setting the `dataTipField` or the `dataTipFunction` properties of the `GanttSheet` object. You can customize the editing tips by setting the `editingTipFunction` property of the `GanttSheet` object.

The following example shows how to use the `dataTipFunction` and `editingTipFunction` properties to specify custom tooltip texts.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;
      import ilog.gantt.ConstraintKind;
      import ilog.core.DataItem;
      import ilog.gantt.ConstraintItem;
      import ilog.gantt.GanttSheet;
      import ilog.gantt.TaskItem;

      import mx.collections.ArrayCollection;
      import mx.formatters.DateFormatter;
      import mx.utils.StringUtil;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Milestone #1", milestone: "true",
          startTime: "1/29/2008 8:0:0", endTime: "1/29/2008 8:0:0"}
      ]);
      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);

      private var dateFormatter:DateFormatter = new DateFormatter();

      private function initApp():void {
        dateFormatter.formatString = "YYYY/MM/DD JJ:NN";
      }

      public function formatTip(item:DataItem):String {
        if (item is TaskItem)
          return formatTaskTip(TaskItem(item));
        else if (item is ConstraintItem)
          return formatConstraintTip(ConstraintItem(item));

        // Should not happen
        return null;
      }

      private function formatTaskTip(item:TaskItem):String {
        if (item.isMilestone)
        {
          return StringUtil.substitute("{0}\n{1}",
                                       item.label,
                                      dateFormatter.format(item.startTime));

        }
        else
        {
          return StringUtil.substitute("{0}\n{1} - {2}",
                                       item.label,
                                       dateFormatter.format(item.startTime),
                                       dateFormatter.format(item.endTime));
        }
      }

      private function formatConstraintTip(item:ConstraintItem):String {
        var fromExtremity:String;
        var toExtremity:String;

        switch (item.kind) {
          case ConstraintKind.END_TO_END:
            fromExtremity = "end";
            toExtremity = "end";
            break;
          case ConstraintKind.END_TO_START:
            fromExtremity = "end";
            toExtremity = "start";
            break;
          case ConstraintKind.START_TO_END:
            fromExtremity = "start";
            toExtremity = "end";
            break;
          case ConstraintKind.START_TO_START:
            fromExtremity = "start";
            toExtremity = "start";
            break;
        }

        var ganttSheet:GanttSheet = taskChart.ganttSheet;
        var fromTaskItem:TaskItem = ganttSheet.itemToTaskItem(item.fromTask);

        var toTaskItem:TaskItem = ganttSheet.itemToTaskItem(item.toTask);

        return StringUtil.substitute("From {0} of: {1}\nTo {2} of: {3}",
                             fromExtremity,
                             fromTaskItem.label,
                             toExtremity,
                             toTaskItem.label);
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  creationComplete="taskChart.dataGrid.expandAll();"
                  >
    <ilog:dataGrid>
      <ilog:GanttDataGrid>
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
    <ilog:ganttSheet>
      <ilog:GanttSheet dataTipFunction="formatTip"
                       editingTipFunction="formatTip"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example, the texts for the data tip and the editing tip are created using the same function. This function returns a text which provides, for a milestone: the name of the task and its start time; for other tasks: the name of the task and the start and end time of the task; for a constraint: the name and extremity of the source task and the name and extremity of the destination task. Dates are formatted using a custom format.

Defining the working and nonworking periods

The Gantt sheet control displays a work grid behind the tasks. The work grid shows the working and nonworking periods. The level of detail on the working and nonworking periods is dynamically adjusted as the zoom factor along the time axis changes.

The working and nonworking periods are defined by the following properties of the `GanttSheet` object.

Working and nonworking period properties

Property	Description
`nonWorkingDays`	The list of days of the week that are nonworking days.
`workingTimes`	The ranges of working time for each day of the week. Each day of the week can have none, one or several ranges of working time.
`nonWorkingRanges`	Additional arbitrary ranges of nonworking time. These ranges are defined as full days or ranges of time of any duration. The nonworking time periods defined by the `nonWorkingRanges` property take precedence over the working and nonworking times defined by the other properties.

For complete reference information, see the `GanttSheet` class.

For information on the rendering of the working and nonworking periods see *Work grid properties*.

Snapping tasks when moving or resizing

The Gantt sheet snaps the start and end time of tasks while they are being moved or resized. By default the Gantt sheet dynamically selects the snapping time precision based on the current zoom factor along the time axis.

You can define a constant snapping time precision by setting the `snappingTimePrecision` property of the Gantt sheet.

In the following example in MXML, the start and end times of tasks are snapped to a 1 day precision.

```
<ilog:GanttSheet snappingTimePrecision=”{{unit: TimeUnit.DAY, steps: 1}}”/>
```

You can also dynamically set the snapping time precision depending on the zoom factor along the time axis, by listening to the changes in the visible time range and adjusting the `snappingTimePrecision` property.

Task chart item renderers

Describes the default item renderers used to display the graphic representation of task and constraint data items in the Gantt sheet, explains how the item renderers get the information they need to display tasks and constraints, and describes the use of custom renderers in the Gantt sheet and in the data grid.

In this section

Default task item renderer in task chart
Describes the default task item renderer.

Default constraint item renderer
Describes the default constraint item renderer.

Task item renderer architecture in task chart
Describes the task item renderer architecture.

Task item renderer layout in task chart
Explains how the position and size of a task item renderer are determined,

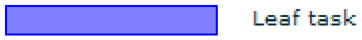
Defining a custom task item renderer for a task chart
Explains how to define a custom task item renderer.

Defining a custom renderer for the data grid of a task chart
Describes how to use custom renderers in the Gantt data grid of a task chart.

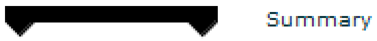
Default task item renderer in task chart

In the Gantt sheet, the default task item renderer displays a bar, start and end symbols, and a label.

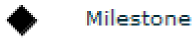
The following figures show examples of the default task item renderer for a leaf task, a summary task and a milestone.



Default task item renderer for a leaf task



Default task item renderer for a summary task



Default task item renderer for a milestone

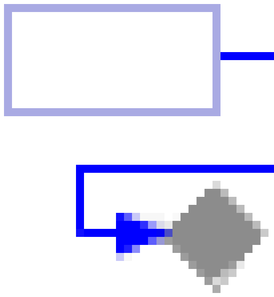
The default task item renderer is defined by the `TaskItemRenderer` class.

You can customize the appearance of the task item renderer using styles and skins. For more information, see *Styling a task chart*.

Default constraint item renderer

The default constraint item renderer displays a link with an end arrow, from the task that is the source of the constraint to the task that is the destination of the constraint.

The following figure shows an example of the default constraint item renderer.



In this example, the constraint links the end of a task to the start of a milestone.

The default constraint item renderer is defined by the `ConstraintItemRenderer` class. You can customize the appearance of the constraint item renderer using skins and styles. For more information see *Constraint styling*.

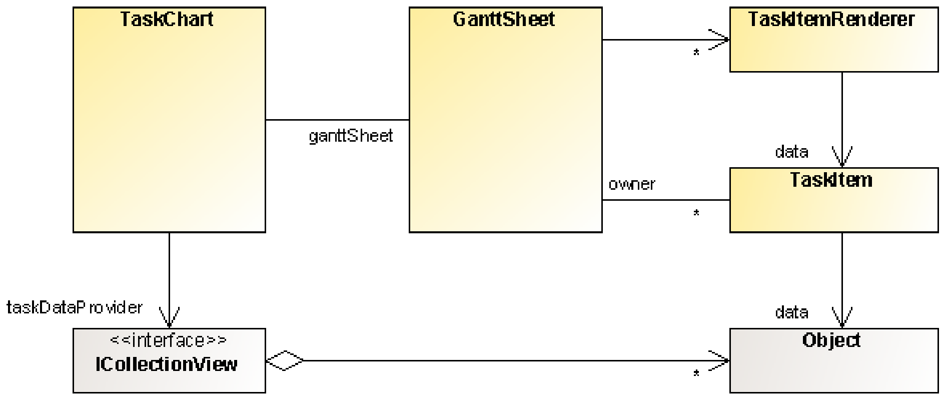
Task item renderer architecture in task chart

The Gantt sheet component uses task item renderers to display the tasks from the task data provider. The same task item renderer is also used to display a task while it is being edited.

For a task item renderer to work with the Gantt sheet component, the Gantt sheet component must be able to pass information to the task item renderer. During editing, the task item renderer must also be able to pass updated information back to the Gantt sheet component.

As with Flex® list-based controls, the Gantt sheet component and the task item renderer use the `data` property to pass information. The contents of the `data` property for a task item renderer are a `TaskItem` object.

The following diagram shows the task item renderer and `TaskItem` object in context.



The `TaskItem` object has `startTime`, `endTime`, `label`, `isSummary`, and `isMilestone` properties that are initialized with the values of the corresponding fields in the data provider item, using the settings defined on the task chart control. It also has a `ganttSheet` property that references the `GanttSheet` object that contains it.

The Gantt sheet component uses the `startTime` and `endTime` properties of the `TaskItem` object to determine the position and size of the renderer.

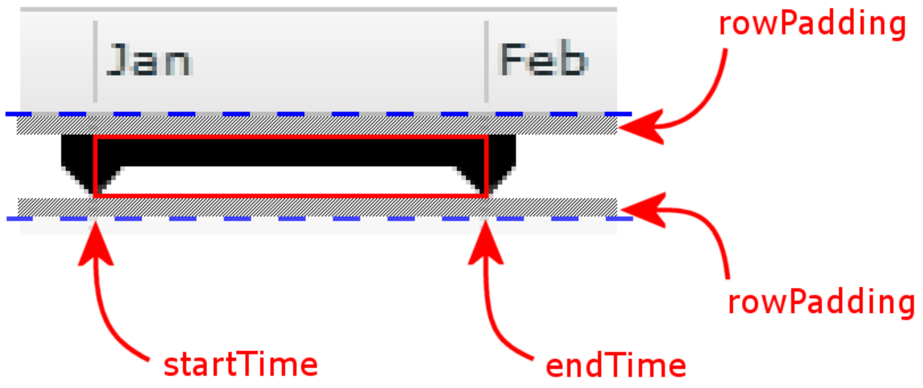
During editing, the Gantt sheet component modifies the `startTime` and `endTime` properties of the `TaskItem` object. The data provider item is updated only at the end of the editing process when the editing is successful.

For more information on the configuration of the `startTime`, `endTime`, `label`, `isSummary`, and `isMilestone` fields see *Configuring task chart data*.

For more information on task editing, see *Editing tasks in a task chart*.

Task item renderer layout in task chart

The following figure shows how the position and size of a task item renderer relates to its containing row, and to the start and end time.



The red rectangle represents the bounds of the task item renderer.

You can also see that:

♦ The task item renderer is vertically padded inside the row by the `rowPadding` style property of the `GanttSheet` object.

♦ On the horizontal axis, the bounds of the task item renderer are defined by the start and end time of the task.

♦ The start and end symbols are drawn beyond the bounds of the task item renderer.

Defining a custom task item renderer for a task chart

To display tasks from the task data provider, the Gantt sheet component uses either the default `TaskItemRenderer` class or the task item renderer specified by the `taskItemRenderer` property.

A custom task item renderer must:

- ♦ Extend the `DisplayObject` class or one of its subclasses. In practice the preferred base classes are `UIComponent`, or `Sprite` for a lighter component.
- ♦ Implement the `IDataRenderer` interface.
- ♦ Implement either the `IInvalidating` interface or the `IProgrammaticSkin` interface.

A custom task item renderer should:

- ♦ Implement the `ISimpleStyleClient` interface in order to use styling.
- ♦ Implement the `ilog.gantt.INode` interface when it can draw beyond its bounds, as defined by the `x`, `y`, `height` and `width` properties.

The x-coordinate and the width of the task item renderer are set by the Gantt sheet depending on the start and end time of the data item. The y-coordinate and the height of the task item renderer are set by the Gantt sheet based on the position and height of the row corresponding to the task in the data grid of the task chart.

The contents of the `data` property for a task item renderer are a `TaskItem` object. The `TaskItem` class allows you to access the `startTime`, `endTime`, `label`, `isSummary` and `isMilestone` fields, the task data provider item, the `GanttSheet` object, and the interaction state of the task item renderer.

You can access the `startTime`, `endTime`, `label`, `isSummary` and `isMilestone` fields inside the task item renderer as follows.

```
   var taskItem:TaskItem = data as TaskItem;
   var startTime:Date = taskItem.startTime;
   var endTime:Date = taskItem.endTime;
   var label:String = taskItem.label;
   var isSummary:Boolean = taskItem.isSummary;
   var isMilestone:Boolean = taskItem.isMilestone;
```

You can access the task data provider item from the task item renderer as follows.

```
   var taskItem:TaskItem = data as TaskItem;
   var dataProviderItem:Object = taskItem.data;
```

Once you have the data provider item and the `TaskItem` object, you can retrieve the GanttSheet and access the interaction state of the task item renderer as follows.

```
var ganttSheet:GanttSheet = taskItem.owner as GanttSheet;
var isHighlighted:Boolean = ganttSheet.isItemHighlighted(dataProviderItem);
var isSelected:Boolean = ganttSheet.isItemSelected(dataProviderItem);
```

The following example in MXML shows how to declare an inline custom task item renderer that displays each task as a box with a completion bar and a label.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0",
          completion: 1.0},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0",
          completion: 0.8},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0",
              completion: 0.3},
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0",
              completion: 0.0 }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet id="ganttSheet">
        <ilog:taskItemRenderer>
          <mx:Component>
            <mx:Canvas horizontalScrollPolicy="off"
                       clipContent="false"
                       backgroundColor="{getBackgroundColor()}"
                       borderColor="0x0000FF"
                       borderStyle="solid"
                       >
              <mx:Script>
                <![CDATA[
                  import ilog.gantt.TaskItem;

                  private function getBackgroundColor():uint {
                    var taskItem:TaskItem = TaskItem(data);
                    if (taskItem.isMilestone || taskItem.isSummary)
                      return 0x000000;
                    else
                      return 0x7777FF;
                  }
                ]]>
              </mx:Script>
              <!-- Completion bar -->
              <mx:Canvas bottom="0"
                         height="30%"
                         percentWidth="{TaskItem(data).data.completion * 100}
"
                         x="1"
                         backgroundColor="0x00FF00"
                         />
              <!-- Label -->
              <mx:Label text="{TaskItem(data).label}"
                        x="{width + 15}"
                        />
            </mx:Canvas>
          </mx:Component>
        </ilog:taskItemRenderer>
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example the background of the task depends on the kind of task: black for milestones and summary tasks, blue for leaf tasks. Tasks have a horizontal green bar showing the completion percentage of the task, mapped from the completion field in the data.

Defining a custom renderer for the data grid of a task chart

As a subclass of `AdvanceDataGrid`, the Gantt data grid leverages the custom renderer functionality provided by its base class.

See *Using item renderers with the AdvancedDataGrid control* in *Adobe Flex 3 Developer's Guide* for information on creating custom renderers for the `AdvancedDataGrid`.

Styling a task chart

Describes how to style the data grid, Gantt sheet, task items, and time scale in a task chart.

In this section

Styles for task charts
Describes the specific styles available for item rendering in a task chart and for the inner components: Gantt sheet, task and constraint items, and time scale.

Data grid styling
Describes the styles available for data grid rendering.

Gantt sheet styling
Describes the style properties that allow you to customize the visual appearance of the Gantt sheet.

Time scale styling
Describes the style properties that allow you to customize the visual appearance of the time scale.

Styling of Gantt sheet items
Describes the style properties that allow you to customize the items in the Gantt sheet.

Task styling for a task chart
Describes the ways that you can style tasks in a task chart when these are rendered using the default task item renderer.

Constraint styling
Describes the ways that you can style constraints when these are rendered using the default constraint item renderer.

Styles for task charts

In addition to the styles provided by its base types, the task chart provides a set of styles that can be used to customize the rendering of its inner components and items:

- `constraintItemStyleName` – the name of the style used by the constraint item renderers
- `dataGridStyleName` – the name of the style used by the Gantt data grid
- `ganttSheetStyleName` – the name of the style used by the Gantt sheet
- `milestoneItemStyleName` – the name of the style used by the task item renderers for milestone tasks
- `summaryItemStyleName` – the name of the style used by the task item renderers for summary tasks
- `taskItemStyleName` – the name of the style used by the task item renderers for leaf tasks
- `timeScaleStyleName` – the name of the style used by the time scale

By default the style name for tasks item renderers depends on whether the task item is a milestone, a summary task or a leaf task. You can override this behavior by providing a custom function in the `itemStyleNameFunction` property of the Gantt sheet.

The following example in MXML shows how you can customize these styles.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myGanttSheetStyle {
      nonWorkingColor: #EEEECC;
      workingColor: #FFFFCC;
    }
    .myDataGridStyle {
      rollOverColor: red;
    }
    .myTimeScaleStyle {
      rollOverColor: red;
    }
    .myConstraintItemStyle {
      linkColor: #00AA00;
      arrowColor: #00AA00;
    }
    .myTaskItemStyle {
      backgroundColor: #70FF70;
      textPosition: right;
      rollOverColor: red;
    }
    .mySummaryItemStyle {
      backgroundColor: #00AA00;
      rollOverColor: red;
      startSymbolColor: #00AA00;
      startSymbolShape: down-pentagon;
      endSymbolColor: #00AA00;
      endSymbolShape: down-pentagon;
      barTopMargin: 0;
      barBottomMargin: 0.5;
      textPosition: right;
    }
    .myMilestoneItemStyle {
      backgroundColor: #00AA00;
      rollOverColor: red;
      startSymbolColor: #00AA00;
      startSymbolShape: diamond;
      barTopMargin: 1;
      barBottomMargin: 1;
      textPosition: right;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" },
        { fromId:"T2", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T5", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
```

```
                  ganttSheetStyleName="myGanttSheetStyle"
                  dataGridStyleName="myDataGridStyle"
                  timeScaleStyleName="myTimeScaleStyle"
                  constraintItemStyleName="myConstraintItemStyle"
                  taskItemStyleName="myTaskItemStyle"
                  summaryItemStyleName="mySummaryItemStyle"
                  milestoneItemStyleName="myMilestoneItemStyle"
                  />
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the renderers of milestones, leaf and summary tasks, constraints, and the inner components of the task chart. The CSS classes `myTaskItemStyle`, `mySummaryItemStyle` and `myMilestoneItemStyle` reproduce the default layout of tasks and customize the default and rollover colors. The other CSS classes customize the rollover color of the data grid and time scale, the color of the work grid in the Gantt sheet, and the color of constraints.

See the `TaskChart` class for more information on these styles.

Data grid styling

The Gantt data grid leverages all the built-in styles provided by `AdvancedDataGrid`, so you can use and set all the styles in the same way as you would with `AdvancedDataGrid`.

See the `AdvancedDataGrid` API in *Adobe Flex 3 Language Reference* for more information on the available styles.

Gantt sheet styling

In addition to the styles provided by its base types the `GanttSheet` control defines its own styles.

The Gantt sheet display is composed of the following elements, arranged from back to front:

♦ Row grid

♦ Work grid

♦ Time grid

♦ Tasks – styling of the tasks is described in the section *Task styling for a task chart*

♦ Constraints – styling of the constraints is described in the section *Constraint styling*

♦ Time indicator

Note: Keep this back-to-front order in mind when defining the color settings of your Gantt sheet. An opaque element in one layer hides the corresponding area in the underlying layers.

Row grid properties

The row grid is a set of horizontal visual elements that aid in associating tasks displayed in the Gantt sheet with their respective resources. The following are the main elements of the row grid, arranged from back to front:

♦ Rows where tasks are displayed. These rows are aligned with the resources corresponding to the tasks in the Gantt data grid. You can customize their styling with the following style property: `alternatingItemColors`.

♦ Horizontal grid lines that delimit the rows. You can customize their styling with the following style property: `horizontalGridLineColor`.

For the horizontal grid lines to be shown, you have to set the `showHorizontalGridLines` property on the Gantt sheet to `true`, as it is not displayed by default.

The following example shows how to customize the row grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" },
        { fromId:"T2", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T5", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet alternatingItemColors="{[0xECF6F4, 0xEDE5F3]}"
                       showHorizontalGridLines="true"
                       horizontalGridLineColor="0xFFFFFF" />
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the alternating item colors of the Gantt sheet and the color of the horizontal grid lines.

Work grid properties

The work grid shows vertical areas that represent the working and nonworking time ranges in the Gantt sheet. You can customize the work grid with the following properties: `nonWorkingAlpha`, `nonWorkingColor`, `nonWorkingFill`, `workingColor`, `workingAlpha`, `workingFill`.

If both `workingColor` and `workingFill` are set, `workingFill` takes precedence. By default, `workingFill` is not set, so `workingColor` is used. The same applies to `nonWorkingColor` and `nonWorkingFill`.

The following example shows how to customize the simpler work grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>
  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet workingColor="green" workingAlpha="0.5"
                       nonWorkingColor="yellow" nonWorkingAlpha="0.5"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the working and nonworking color and transparency of the work grid in the Gantt sheet.

The work grid can also be customized to use fill styles to paint the working and nonworking vertical areas. You can customize the fill styles in two ways:

1. In ActionScript® , retrieve the instance of `GanttSheet` and set its style properties with the appropriate instances of `IFill`.
2. In MXML, set the `ganttSheet` property of `TaskChart` with a `GanttSheet` instance that has been customized with your fill styles.

The following example shows the MXML way.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet>
        <ilog:workingFill>
          <mx:LinearGradient>
            <mx:entries>
              <mx:GradientEntry color="red" ratio="0" alpha="0.5"/>
              <mx:GradientEntry color="blue" ratio="0.33" alpha="0.5"/>
            </mx:entries>
          </mx:LinearGradient>
        </ilog:workingFill>
        <ilog:nonWorkingFill>
          <mx:SolidColor color="yellow" alpha="0.4"/>
        </ilog:nonWorkingFill>
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the working and nonworking fill styles used to paint the corresponding areas of the Gantt sheet.

Time grid properties

The time grid shows vertical lines in the Gantt sheet that are aligned with the ticks in the minor row of the time scale. You can customize the styling of the time grid with the following properties: `timeGridAlpha` and `timeGridColor`.

The following example shows how to customize the time grid styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet timeGridColor="green" timeGridAlpha="0.9"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the color and transparency of the time grid in the Gantt sheet.

Time indicator properties

The current time indicator shows a vertical line corresponding to the current time in the Gantt sheet. You can customize the styling of the time indicator with the following properties: `currentTimeIndicatorAlpha` and `currentTimeIndicatorColor`.

For the current time indicator to be shown, you have to set the `showCurrentTimeIndicator` property on the Gantt sheet to `true`, as it is not displayed by default.

The following example shows how to customize the time indicator styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import ilog.utils.TimeUnit;
      import ilog.utils.GregorianCalendar;
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);

      [Bindable]
      private var start:Date;

      [Bindable]
      private var end:Date;

      private function initApp():void {
        // Make sure the current time is visible
        var calendar:GregorianCalendar = ganttSheet.calendar;
        var now:Date = new Date();
        var start:Date = calendar.addUnits(now, TimeUnit.WEEK, -1);
        var end:Date = calendar.addUnits(now, TimeUnit.WEEK, 1);
        ganttSheet.visibleTimeRangeStart = start;
        ganttSheet.visibleTimeRangeEnd = end;
      }
    ]]>
  </mx:Script>
  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet id="ganttSheet"
                       showCurrentTimeIndicator="true"
                       currentTimeIndicatorColor="blue"
                       currentTimeIndicatorAlpha="0.8"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the color and transparency of the current time indicator in the Gantt sheet.

Animation properties

The animation properties are used to control the animation aspects of the Gantt sheet, namely:

- ♦ The animation duration that determines how long the animations will last
- ♦ The function used to determine the pace of progression of an animation

For more details on the available animation features, see *Animating a task chart*.

The following example shows how to customize the animation-related styles provided by the Gantt sheet.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import mx.effects.easing.Linear;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet animationDuration="200"
                       easingFunction="{Linear.easeNone}"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the Gantt sheet’s animation duration and the easing function.

To disable time navigation animation altogether, just set the `animationDuration` style property of the Gantt sheet to zero as follows:

```
     <ilog:GanttSheet animationDuration="0"/>
```

Time scale styling

In addition to the styles provided by its base types the time scale control defines its own styles.

The time scale display is composed of the following elements:

- ♦ Background
- ♦ The time scale rows and separator
- ♦ Interaction feedback

There are two time scale rows: the major time scale row at the top and the minor time scale row at the bottom. Each time scale row is composed of ticks and text that do not overlap.

See also the `TimeScale` class for more information on the available styles.

Background properties

The background is rectangle that is drawn behind all elements of the time scale. Its style can be customized with the following property: `backgroundColors`.

The `backgroundColors` property takes an array with two colors. If the colors provided are the same, the background is painted with the specified color. Otherwise, the background is painted using a gradient that goes from the first color at the top of the time scale to the second color at the bottom.

The following example shows how to customize the background styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>
  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:timeScale>
      <ilog:TimeScale backgroundColors="{['white', 'black']}"/>
    </ilog:timeScale>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the background colors of the time scale.

Separator properties

The separator is displayed by the time scale between the major and minor time scale rows. Its style can be customized with the following properties: `separatorAlpha`, `separatorColor` and `separatorThickness`.

The following example shows how you can easily customize the separator styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      public var resources:ArrayCollection = new ArrayCollection([
        { id: "R1", name: "Project Manager", location: "Geneva" }
      ]);

      [Bindable]
      public var tasks:ArrayCollection = new ArrayCollection([
        { resourceId: "R1", name: "Define project vision",
          startTime: "1/14/2008 8:0:0", endTime: "1/15/2008 17:0:0" },
        { resourceId: "R1", name: "Refine project vision",
          startTime: "1/16/2008 8:0:0", endTime: "1/17/2008 17:0:0" },
        { resourceId: "R1", name: "Assess project vision",
          startTime: "1/18/2008 8:0:0", endTime: "1/21/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:ResourceChart id="resourceChart" width="100%" height="100%"
                      resourceDataProvider="{resources}"
                      taskDataProvider="{tasks}">
    <ilog:timeScale>
      <ilog:TimeScale separatorAlpha="0.5"
                      separatorColor="red"
                      separatorThickness="2"/>
    </ilog:timeScale>
  </ilog:ResourceChart>
</mx:Application>
```

This example customizes the color, transparency and thickness of the separator of the time scale.

Tick properties

The ticks separate the time segments in both the major and minor time scale rows. You can customize the styling of ticks with the following properties: `majorTickAlpha`, `majorTickColor`, and `minorTickAlpha`.

The following example shows how to customize the tick styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:timeScale>
      <ilog:TimeScale majorTickAlpha="0.5" majorTickColor="red"
                      minorTickAlpha="0.5" minorTickColor="blue"/>
    </ilog:timeScale>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the time scale’s major and minor ticks color and transparency.

Time scale text properties

The text contains a label for each time interval shown by the time scale rows in the time scale. You can customize the styling of text with the following properties: `majorTextStyleName` and `minorTextStyleName`.

The following example shows how to customize the text styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myMajorTextStyle {
      color: blue;
      fontWeight: bold;
    }
    .myMinorTextStyle {
      color: red;
      fontStyle: italic;
    }
  </mx:Style>
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:timeScale>
      <ilog:TimeScale majorTextStyleName="myMajorTextStyle"
                      minorTextStyleName="myMinorTextStyle"/>
    </ilog:timeScale>
  </ilog:TaskChart>
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the text of the major and minor rows in the time scale. The CSS classes customize the color, font style and weight of the text.

Highlighting properties

Highlighting comprises visual elements that aid in providing feedback to users who are interacting with the time scale. The following are the main highlighting visual elements:

- ♦ Highlighting of predefined time intervals when users hover over them on the time scale
- ♦ Highlighting of arbitrary time intervals when users select them on the time scale

The highlighting style can be customized with the following properties: `rollOverAlpha` and `rollOverColor`.

The following example shows how to customize the highlighting styles provided by the time scale.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:timeScale>
      <ilog:TimeScale rollOverAlpha="0.5" rollOverColor="red"/>
    </ilog:timeScale>
  </ilog:TaskChart>
</mx:Application>
```

This example customizes the rollover color and transparency of the time scale.

Styling of Gantt sheet items

The Gantt sheet assigns a style to task item renderers and constraint item renderers depending on the kind of item that is represented. The following table lists the kinds of items, the style name assigned by default, and the `TaskChart` property that defines the value.

Kind of item	Default style name	Property
constraint	`undefined`	`constraintItemStyleName`
leaf task	`leafTask`	`taskItemStyleName`
milestone task	`milestoneTask`	`milestoneItemStyleName`
summary task	`summaryTask`	`summaryItemStyleName`

You can provide a custom function to override the default mapping from the kind of object to the style name, using the item style name function specified in the `itemStyleNameFunction` property.

For more information on styling tasks see *Task styling for a task chart*.

For more information on styling constraints see *Constraint styling*.

Task styling for a task chart

The task display for the default task item renderer is composed of the following elements:

- Bar
- Start symbol
- End symbol
- Text
- Roll over and selection feedback

The `TaskItemRenderer` uses skins as shown in the following table.

Property name	Default value	Description
`barSkin`	`ilog.gantt.TaskBarSkin`	The skin used to render the task bar.
`startSymbolSkin`	`ilog.gantt.TaskSymbolSkin`	The skin used to render the start symbol of a task.
`endSymbolSkin`	`ilog.gantt.TaskSymbolSkin`	The skin used to render the end symbol of a task.

See the `TaskItemRenderer` API for more information on the available style properties.

Bar properties

The bar is a rectangle that is drawn from the start time to the end time of the task. The top and bottom margins allow you to position the bar vertically within the task item renderer. The margins are defined by the properties `barTopMargin` and `barBottomMargin`.

The colors of the bar and border are defined by the `backgroundColor` and `borderColor` properties. The width of the border is defined by the `borderThickness` property.

To hide the bar, set `barTopMargin` to 1.0 and `barBottomMargin` to 1.0.

The following example shows how to customize the background and border colors provided by the default task item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myTaskItemStyle {
      backgroundColor : #FFCCCC;
      borderColor : #FF0000;
      borderThickness : 2.0;
      textPosition: right;
    }
    .mySummaryItemStyle {
      backgroundColor : #FFCCCC;
      borderColor : #FF0000;
      borderThickness : 2.0;
      startSymbolShape: down-pentagon;
      endSymbolShape: down-pentagon;
      barTopMargin: 0;
      barBottomMargin: 0.5;
      textPosition: right;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                      taskDataProvider="{tasks}"
                      taskItemStyleName="myTaskItemStyle"
                      summaryItemStyleName="mySummaryItemStyle"/>
</mx:Application>
```

This example specifies the name of the CSS classes to be used to style the task item renderers for leaf and summary tasks. The styles for these CSS classes reproduce the default layout of leaf and summary tasks and customize the width of the border and the colors of the bars for leaf and summary tasks.

Start and end symbol properties

The start and end symbols mark the start and end time of a task. The start symbol is displayed centered at the start time of a task. Both its width and its height are equal to the height of the task item renderer. The end symbol is displayed centered at the end time of a task.

The default skin for symbols provides different shapes. Use the `startSymbolShape` property and `endSymbolShape` property to select the shape of the start and end symbols respectively.

Shape	Name
- (hidden)	`none`
⬟	`upPentagon`
⛊	`downPentagon`
◆	`diamond`
▲	`upTriangle`
▼	`downTriangle`
⇧	`upArrow`
⇩	`downArrow`

To hide the start symbol or end symbol, set the `startSymbolShape` or `endSymbolShape` to `none`.

The rendering of the start symbol can be customized with the following properties: `startSymbolBorderColor`, `startSymbolBorderThickness`, `startSymbolColor`, `startSymbolShape`, and `startSymbolSkin`. For the end symbol, the corresponding properties are: `endSymbolBorderColor`, `endSymbolBorderThickness`, `endSymbolColor`, `endSymbolShape`, and `endSymbolSkin`.

The following sample shows how to customize the start and end symbols of the default task item renderer for summary tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .mySummaryItemStyle {
      startSymbolColor: #FF0000;
      startSymbolBorderColor: #880000;
      startSymbolShape: up-triangle;
      endSymbolColor: #FF0000;
      endSymbolBorderColor: #880000;
      endSymbolShape: down-triangle;
      barTopMargin: 0;
      barBottomMargin: 0.5;
      textPosition: right;
    }
    .myMilestoneItemStyle {
      startSymbolColor: #FF0000;
      startSymbolBorderColor: #880000;
      startSymbolShape: down-arrow;
      barTopMargin: 1;
      barBottomMargin: 1;
      textPosition: right;
    }
  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  summaryItemStyleName="mySummaryItemStyle"
                  milestoneItemStyleName="myMilestoneItemStyle"
                  />
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the task item renderers for summary tasks and milestones. The styles for these CSS classes reproduce the default layout of summary tasks and milestones and customize the shape and colors of the start and end symbols of summary tasks as well as the start symbol for milestones.

Task text properties

The text is a label that provides a written description for a task. You can customize the styling of the text with the `textStyleName` and `textPosition` properties.

The following example shows how to customize the text style provided by the default task item renderer.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Style>
    .myTaskItemStyle {
      textStyleName: "myTextStyle";
      textRollOverColor: red;
      textSelectedColor: green;
      textSelectedRollOverColor: black;
      textPosition: left;
    }
    .mySummaryItemStyle {
      textStyleName: "myTextStyle";
      textRollOverColor: red;
      textSelectedColor: green;
      textSelectedRollOverColor: black;
      textPosition: left;
      startSymbolShape: down-pentagon;
      endSymbolShape: down-pentagon;
      barTopMargin: 0;
      barBottomMargin: 0.5;
    }
    .myMilestoneItemStyle {
      textStyleName: "myTextStyle";
      textRollOverColor: red;
      textSelectedColor: green;
      textSelectedRollOverColor: black;
      textPosition: left;
      startSymbolShape: diamond;
      barTopMargin: 1;
      barBottomMargin: 1;
    }
    .myTextStyle {
      color: blue;
      fontWeight: bold;
    }

  </mx:Style>

  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  taskItemStyleName="myTaskItemStyle"
                  milestoneItemStyleName="myMilestoneItemStyle"
                  summaryItemStyleName="mySummaryItemStyle"/>
</mx:Application>
```

This example specifies the names of the CSS classes to be used to style the task item renderers for leaf and summary tasks and milestones. The styles for these CSS classes reproduce the default layout of these kinds of tasks, and customize the text color and the position of the task labels.

Roll over and selection feedback

The default task item renderer allows you to customize the colors of the task bar, symbols and text for the interaction states: selected, rolled over, selected and rolled over.

The following table shows the style properties used for coloring the elements of the task item renderer depending on the selection state.

Element	Default	Selected	Rolled over	Selected and rolled over
bar background	`backgroundColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
start symbol background	`startSymbolColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
end symbol background	`endSymbolColor`	`selectedColor`	`rollOverColor`	`selectedRollOverColor`
bar border	`borderColor`	`borderSelectedColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
start symbol border	`startSymbolBorderColor`	`startSymbolBorderColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
end symbol border	`endSymbolBorderColor`	`endSymbolBorderColor`	`borderRollOverColor`	`borderSelectedRollOverColor`
text	`color`	`textSelectedColor`	`textRollOverColor`	`textSelectedRollOverColor`

Constraint styling

The constraint display for the default constraint item renderer is composed of the following elements:

- Link
- Arrow

See the `ConstraintItemRenderer` class for more information on the available style properties.

Constraint link properties

The constraint link represents the constraint between two tasks. You can customize the way it is rendered with the following properties: `linkAlpha`, `linkColor`, `linkStroke`, `linkThickness`.

Constraint arrow properties

The constraint arrow points to the task that is the destination (target) of the constraint. You can customize the way it is rendered with the following properties: `arrowSize`, `arrowAlpha`, `arrowColor`, `arrowStroke`, `arrowBorderThickness`.

Animating a task chart

Describes how to use the built-in animations in a task chart.

In this section

Time navigation
Describes how to use the built-in animation when navigating in time.

Expand and Collapse
Describes how to use the built-in animations when expanding and collapsing tasks.

Time navigation

The Gantt sheet and the time scale provide built-in support for animating operations related to time navigation. For example, when users zoom in or out, pan to a given time, or focus on a given time range, all of these operations can be animated.

Note: By default, all of the built-in interactions provided by the Gantt sheet and time scale are animated.

The animation settings for time navigation are held by the Gantt sheet. They control both the animation of time navigation in the Gantt sheet and animation in the time scale.

For more details of the available animation-related styles, see *Animation properties*.

The `GanttSheet` component also allows you to animate the following time navigation operations through its API by setting the `animate` parameter.

```
moveTo(time:Date, animate:Boolean = false):void
showAll(margin:Number = 10, animate:Boolean = false):void
zoom(ratio:Number, time:Date = null, animate:Boolean = false):void
```

Expand and Collapse

The Gantt sheet leverages the built-in support of the Gantt data grid for animating the expanding and collapsing of items, and provides a similar animation for its tasks. It synchronizes the position and visibility of tasks with the position and visibility of the corresponding resources when they are expanded or collapsed in the Gantt data grid either by user interaction or through the API.

For information on the built-in animation support leveraged by the Gantt data grid from the Adobe® `AdvancedDataGrid` class, see the Adobe documentation for `AdvancedDataGrid`.

Managing task chart events

Describes how to manage events that are a result of user interactions with a task chart.

In this section

Data grid events
Describes how to manage events that are a result of user interactions with the data grid in a task chart.

Gantt sheet data events
Describes how to manage events that are a result of user interactions with the data on a Gantt sheet in a task chart.

Gantt sheet navigation events
Describes how to manage events that are a result of users navigating in the Gantt sheet in a task chart.

Data grid events

The Gantt data grid leverages all the event types provided by the `AdvancedDataGrid`. The `AdvancedDataGrid` generates several event types that let you respond to user interaction on its displayed rows.

See *Handling events in a DataGrid control* in *Adobe Flex 3 Developer's Guide* and *AdvancedDataGridEvent* in the *Adobe Flex 3 Language Reference* for more information on the available events.

Handling task selection

If you need to process events triggered by the selection of tasks of the Gantt data grid, you can use the `AdvancedDataGrid` API. The following example shows how to process events in this way.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import mx.controls.Alert;
      import mx.events.ListEvent;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);

      private function handleDataGridChange(event:ListEvent):void {
        Alert.show(event.itemRenderer.data['name'] + " was selected!",
                   "Change Event");
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:dataGrid>
      <ilog:GanttDataGrid change="handleDataGridChange(event)" />
    </ilog:dataGrid>
  </ilog:TaskChart>
</mx:Application>
```

This example uses `ListEvent.CHANGE` triggered by the Gantt data grid to notify when a resource is selected. This basic example shows a message to the user with the name of the resource that was selected.

Expanding task rows

If you need to expand or collapse tasks of the Gantt data grid programmatically, you can use the `AdvancedDataGrid` API. The following example shows how to expand or collapse tasks programmatically.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"},
        { id: "T3", name: "Summary Task #1",
          startTime: "2/5/2008 8:0:0", endTime: "2/17/2008 17:0:0",
          children: [
            { id: "T4", name: "Task #3",
              startTime: "2/5/2008 8:0:0", endTime: "2/12/2008 17:0:0" },
            { id: "T5", name: "Task #4",
              startTime: "2/13/2008 8:0:0", endTime: "2/17/2008 17:0:0" }
          ]},
        { id: "T6", name: "Milestone #1", milestone: "true",
          startTime: "2/17/2008 17:0:0", endTime: "2/17/2008 17:0:0" }
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" },
        { fromId:"T2", toId:"T4", kind:"endToStart" },
        { fromId:"T4", toId:"T5", kind:"endToStart" },
        { fromId:"T5", toId:"T6", kind:"endToStart" }
      ]);

      public var summaryTask1:Object = tasks.source[2];
      public var expand:Boolean = true;
      private function expandCollapseRow():void
      {
        taskChart.dataGrid.expandItem(summaryTask1, expand, true);
        expand = !expand;
      }
    ]]>
  </mx:Script>

  <mx:Button label="Expand/Collapse Summary Task #1"
             click="expandCollapseRow()"/>
  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                      taskDataProvider="{tasks}"
                      constraintDataProvider="{constraints}">
  </ilog:TaskChart>
</mx:Application>
```

Gantt sheet data events

A GanttSheet component fires several types of events in response to user interactions with the data. You can create an event handler for each of these events.

You can listen for the following GanttSheet data events.

Gantt sheet data events

GanttSheet data event	Description
change	Indicates that the selection changed as a result of user interaction.
itemClick	Indicates that the user clicked the pointer over a visual item in the control.
itemDoubleClick	Indicates that the user double-clicked the pointer over a visual item in the control.
itemEditBegin	Indicates that the user started editing an item, either by dragging one of its extremities or by dragging it.
itemEditEnd	Indicates that the editing of an item has completed.
itemEditMove	Indicates that the user is moving an item.
itemEditResize	Indicates that the user is resizing an item.
itemRollOut	Indicates that the user rolled the pointer out of a visual item in the control.
itemRollOver	Indicates that the user rolled the pointer over a visual item in the control.

The Gantt sheet data events are of type GanttSheetEvent.

Within an event handler for a Gantt sheet data event you can access the data provider item associated with the event. You can also access the corresponding task item renderer and its TaskItem object as well as the GanttSheet object.

For information on using the task item editing events see *Using task editing events*.

In the following example a popup is displayed whenever a task is clicked, retrieving the label of the task from the TaskItem object.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import mx.controls.Alert;
      import ilog.gantt.ConstraintItem;
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);

      public function onItemClick(event:GanttSheetEvent):void {
        var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
        if (taskItem)
        {
          Alert.show("You have clicked on the task:\n" + taskItem.label,
                     "Click Event");
          return;
        }

        var constraintItem:ConstraintItem =
            event.itemRenderer.data as ConstraintItem;

        if (constraintItem)
        {
          var fromTask:Object = constraintItem.fromTask;
          var toTask:Object = constraintItem.toTask;
          Alert.show("You have clicked on the constraint:\n"
                     + "kind: " + constraintItem.kind + "\n"
                     + "from: " + ganttSheet.itemToTaskItem(fromTask).label
                     + "\n"
                     + "to: " + ganttSheet.itemToTaskItem(toTask).label,
                     "Click Event");
          return;
        }
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet id="ganttSheet"
                       itemClick="onItemClick(event);">
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

The previous example uses the `TaskItem` object to retrieve the label of the task. You can achieve the same result by retrieving the label directly from the `name` property of the data provider item.

```
    public function onItemClick(event:GanttSheetEvent):void {
       Alert.show("You have clicked on the task:\n" + event.item.name,
                  "Click Event");
     }
```

For more information on the `TaskItem` class see *Task item renderer architecture in task chart*.

Gantt sheet navigation events

The Gantt sheet control fires an event of type `visibleTimeRangeChange` when the visible time range is modified, typically as a result of a user interaction.

Gantt sheet navigation event

GanttSheet navigation event	Description
`visibleTimeRangeChange`	Indicates that the visible time range has changed.

The Gantt sheet visible time range event is of type `GanttSheetEvent`.

During animation of the change in the visible time range, a listener receives a `visibleTimeRangeChange` event for each animation step. You can use the `adjusting` property of the `GanttSheetEvent` object to determine whether the event is occurring during an animation. Each intermediate event has the `adjusting` property set to `true`; this property is set to `false` for the last event of an animation or when the change is not animated.

The following example shows how to display the visible time range whenever it is modified.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;
      import mx.controls.Alert;
      import ilog.gantt.GanttSheetEvent;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"}
      ]);

      public function onVisibleTimeRangeChange(
                                  event:GanttSheetEvent):void {

        // Ignore the intermediate events generated during animation or
        // during user interactions.
        if (event.adjusting)
          return;

        Alert.show("The visible time range is now:\n" +
           "from " + GanttSheet(event.target).visibleTimeRangeStart+
           "\nto " + GanttSheet(event.target).visibleTimeRangeEnd,
           "VisibleTimeRangeChange Event");
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  >
    <ilog:ganttSheet>
      <ilog:GanttSheet
        visibleTimeRangeChange="onVisibleTimeRangeChange(event)"/>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example, if the `visibleTimeRangeChange` event is dispatched during animation the event is ignored.

User interaction with task charts

Describes possible user interactions with task charts.

In this section

User interaction in the Gantt data grid
Describes possible user interactions with the data grid in a task chart.

User interaction in the time scale
Describes the facilities for navigation in time and control of the time range.

User interaction in the Gantt sheet
Describes the facilities for navigation in the Gantt sheet and for interacting with tasks.

User interaction in the Gantt data grid

The user interaction support available in the Gantt data grid, such as selection, sorting, editing, and the drag-and-drop feature, is provided entirely by the built-in behavior of `AdvancedDataGrid`.

See *DataGrid control user interaction* and *Sorting data in DataGrid controls* in *Adobe Flex 3 Developer's Guide* for more information on the possible built-in user interactions of `AdvancedDataGrid`.

Task editing in the data grid

The following example shows how to enable the editing of tasks in the Gantt data grid by using its `editable` property.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0" },
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0" }
      ]);
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}">
    <ilog:dataGrid>
      <ilog:GanttDataGrid editable="true">
        <ilog:columns>
          <mx:AdvancedDataGridColumn dataField="id"
                                     headerText="ID"
                                     editable="false"/>
          <mx:AdvancedDataGridColumn dataField="name"
                                     headerText="Name"/>
          <mx:AdvancedDataGridColumn dataField="startTime"
                                     headerText="Start"/>
          <mx:AdvancedDataGridColumn dataField="endTime"
                                     headerText="End"/>
        </ilog:columns>
      </ilog:GanttDataGrid>
    </ilog:dataGrid>
  </ilog:TaskChart>
</mx:Application>
```

This example sets the `editable` property of the Gantt data grid to `true` to enable its editing features. The example also displays columns `ID` , `Name`, `Start`, and `End`. The `Name`, `Start`,and `End` columns are editable and the `ID` column is read-only. These characteristics are achieved by setting the `editable` property of each column appropriately.

User interaction in the time scale

The time scale allows users to navigate in time and to control the time range displayed by the task chart by using standard user interactions such as mouse clicks and dragging objects with the mouse.

The `GanttSheet` control broadcasts `visibleTimeRangeChange` events whenever the visible time range is modified.

Mouse navigation

The time scale has the mouse navigation features shown in the following table.

Mouse Navigation features in the time scale

Action	Description of action	Mouse or key and mouse action
Highlight	Highlights the hovered time interval.	Move.
Focus	Focuses on the selected time interval and makes it entirely visible.	Click.
Scroll	Scrolls the visible time range forward or backward in time.	Drag.
Focus interval	Focuses on an arbitrary time interval and makes it entirely visible.	Hold down the CTRL key and drag.
Zoom	Zooms the visible time in or out. The time under the mouse pointer is the same before and after zooming.	Hold down the CTRL key and rotate the mouse wheel forward or backward.

Note:
1. Navigation is limited by the `minVisibleTime` or `maxVisibleTime` value.
2. Zooming is limited by the `minZoomFactor` or `maxZoomFactor` value and the `minVisibleTime` or `maxVisibleTime` value, depending on which is the most constraining.

Attempting to zoom or navigate beyond these limits has no effect on the task chart.

The time scale has the keyboard navigation features shown in the following table.

Keyboard navigation features in the time scale

Interaction	Description
+	Zooms the visible time in.
-	Zooms the visible time out.
Keyboard arrows	Lets the user move up, down, or from side to side in the Gantt sheet

User interaction in the Gantt sheet

The Gantt sheet and tasks respond to mouse gestures to provide end users with interactions.

Interactions with the Gantt sheet

The following table shows the mouse interactions available in the Gantt sheet.

Mouse interactions in the Gantt sheet

Interaction	Action
Hold down the CTRL key and rotate the mouse wheel forward or backward	Zooms the visible time in or out. The time under the mouse pointer is the same before and after zooming.
Drag	Lets the user move up, down, or from side to side in the Gantt sheet.

The `GanttSheet` control broadcasts `visibleTimeRangeChange` events when the visible time range is modified by user interaction.

The Gantt sheet has the mouse navigation features shown in the following table.

Keyboard navigation features in the Gantt sheet

Interaction	Description
+	Zooms the visible time in.
-	Zooms the visible time out.
Keyboard arrows	Lets the user move up, down, or from side to side in the Gantt sheet

Interactions with tasks

The `GanttSheet` control broadcasts:

- `change` events when the selection is modified by a user interaction.
- `itemEditBegin`, `itemEditMove`, `itemEditResize`, and `itemEditEnd` events when the user moves or resizes tasks.

The following table shows the mouse interactions available on tasks.

Mouse interactions on tasks

Interaction	Action
Hover	Highlights the task.
Click	Selects the task.
Drag the task horizontally	Moves the tasks along the time axis. The start and end time are adjusted.
Drag the start or end of the task bar	Changes the start or end time of the task.
Hold down the CTRL key and click	Extends or reduces the selection by toggling the selection state of the task that has been clicked. You must set the `allowMultipleSelection` property to `true` to allow multiple selection.

Interaction with constraints

The `GanttSheet` control broadcasts `change` events when the selection is modified by a user interaction.

The following table shows the mouse interactions available on constraints.

Interaction	Action
Hover	Highlights the constraint.
Click	Selects the constraint.
Hold down the CTRL key and click	Extends or reduces the selection by toggling the selection state of the constraint that has been clicked. You must set the `allowMultipleSelection` property to `true` to allow multiple selection.

Editing tasks in a task chart

Describes the task editing process and how to use events associated with task editing when the user moves or resizes tasks in the task chart control by direct manipulation.

In this section

About task editing in a task chart
Discusses task editing in general.

Task editing process
Gives the sequence of steps when a task is edited.

Using task editing events
Describes the editing events and how to use them to customize editing.

Updating the data provider item
Explains how to update the data provider item after a task edit.

Accessing task data and the task item renderer in an event listener
Describes how a Gantt sheet event listener can access task data and the task item renderer.

Determining the kind of edit in an event listener
Describes how an event listener can determine the kind of editing action in progress.

Determining the reason for an itemEditEnd event
Describes how to determine the reason for an event signaling the end of an edit session.

Examples of editing event handlers
Gives several examples demonstrating the use of editing events.

About task editing in a task chart

Editing tasks in a Gantt sheet has some similarities with editing cells in Flex list-based components. However there are also differences: there is no item editor when you edit tasks and there is no `itemEditBeginning` event; you receive intermediate events for each change while editing a task.

The `moveEnabled` and `resizeEnabled` properties of the Gantt sheet control whether the user can move and resize the tasks. The `moveEnabledFunction` and `resizeEnabledFunction` properties specify a custom function (if any) to control the editing capabilities for a specific task.

To represent the task being edited during editing, the Gantt sheet uses either the default `TaskItemRenderer` object or the task item renderer specified by the `taskItemRenderer` property.

For more information on the task item renderer see *Default task item renderer in task chart*.

Task editing process

The following sequence of steps occurs when a task is edited in the Gantt sheet:

1. The user holds down the mouse button on a task and starts dragging.

   The kind of editing operation performed depends on which area of the task is being dragged:

   - The start extremity – when dragged, resizes the task by modifying its start time.
   - The end extremity – when dragged, resizes the task by modifying its end time.
   - The body of the task – when dragged, moves the task along the time axis.

2. The Gantt sheet dispatches the `itemEditBegin` event and shows the editing tooltip. You can use this event to prepare data for editing.

   The next two steps can be repeated any number of times while the user is dragging.

3. The user moves or resizes the task.
4. The Gantt sheet dispatches the `itemEditMove` or `itemEditResize` event. You can use these events to modify the task that is being edited. For more information, see *Using task editing events*.
5. The user ends the editing session. Typically the task editing session ends when the user releases the mouse button.
6. The Gantt sheet dispatches the `itemEditEnd` event to update the data item and the Gantt sheet. You can use this event to modify the task item before it is saved.
7. The task that was being edited is updated in the Gantt sheet.

Using task editing events

A Gantt sheet component dispatches the following events as part of the task editing process: the `itemEditBegin`, `itemEditMove`, `itemEditResize` and `itemEditEnd` events. The Gantt sheet control defines default event listeners for all of these events except `itemEditBegin`.

You can write your own event listeners for one or more of these events to customize the editing process. When you write your own event listener, it executes before the default event listener defined by the component; the default listener executes afterward.

You can also replace the default event listener for the component with your own event listener. To prevent the default event listener from executing, you call the `preventDefault()` method from anywhere in your event listener.

Use the following events when you want to customize the editing of tasks:

itemEditBegin

Dispatched when the editing session starts. The user has pressed the mouse button over a task and started dragging.

The `GanttSheet.editKind` property is set with the editing operation being performed:

- ♦ `TaskItemEditKind.MOVE_REASSIGN` – the user is dragging the task body
- ♦ `TaskItemEditKind.RESIZE_START` – the user is dragging the start extremity of the task
- ♦ `TaskItemEditKind.RESIZE_END` – the user is dragging the end extremity of the task

The Gantt sheet component does not have a default listener for the `itemEditBegin` event. After the event is sent the Gantt sheet displays the editing tool tip which will provide information on the task while it is being edited. For more information see *Defining the tooltips for tasks and constraints*.

You can write an event listener for this event to modify the `TaskItem` data used by the task item renderer, or to modify some other information that will be used during editing.

itemEditMove

Dispatched when the task is being moved along the time axis. The Gantt sheet component has a default listener for the `itemEditMove` event that adjusts the start and end time of the `TaskItem`.

The default event listener performs the following actions:

- ♦ Rounds the start time to the precision defined by the `snappingTimePrecision` property of the `GanttSheet` object.
- ♦ Updates the end time so the duration of the task remains constant.

You can write an event listener for this event to examine and modify the `TaskItem` data used by the task item renderer. For example, you can adjust the start and end time to take into account nonworking periods and maintain a constant work duration for the task.

You can also implement a custom snapping policy or implement specific constraints on the start and end time. In this case you need to call `preventDefault()` to stop the Gantt sheet component from applying its own snapping policy or overriding your adjustments to the start and end time. For more information see *Custom snapping when moving tasks*.

itemEditResize

Dispatched when the task is being resized.

The Gantt sheet component has a default listener for the `itemEditResize` event that adjusts the start or end time of the `TaskItem` object.

The default event listener performs the following actions:

- ♦ Uses the `snappingTimePrecision` property of the Gantt sheet to round the start or end time, depending on the kind of edit operation.
- ♦ Makes sure that the start time is before the end time.

You can write an event listener for this event to modify the `TaskItem` data used by the task item renderer. For example, you can adjust the start or end time to enforce a minimum duration; or you can implement a custom snapping policy; or you can implement other specific constraints on the time values. In these cases you need to call `preventDefault()` to stop the Gantt sheet from applying its own snapping policy or overriding your adjustments to the start and end time. For more information, see *Examples of editing event handlers*

itemEditEnd

Dispatched when the task editing session ends, typically when the user releases the mouse button.

The Gantt sheet component has a default listener for this event that updates the item in the task data provider with the values from the `TaskItem` object used by the task item renderer.

The default event listener performs the following actions:

- ♦ When the edit operation was not cancelled the default event listener calls the `commitItem()` method to update the data provider item from the task item.

After all listeners of `itemEditEnd` have been called, the Gantt sheet component forces an update of the task by calling `ICollectionView.itemUpdated()` for this task data provider item, refreshing the Gantt sheet.

Updating the data provider item

To save changes made in the Gantt sheet back to the data provider, use the `commitItem` method. This method uses the `taskStartTimeField` and `taskEndTimeField` properties of the `TaskChart` to determine which properties of the data provider item must be updated, and stores the edited start and end time in these fields. This behavior applies even if the `taskStartTimeFunction` or `taskEndTimeFunction` properties of the `TaskChart` are defined.

When the data provider item is an XML object, the values of `startTime` and `endTime` are stored as Strings using the `Date.toString` method. When the data provider item is an ActionScript® object, the values of the `startTime` and `endTime` are stored as `Date` objects.

You can override this behavior by providing a custom function and setting the `commitItemFunction` field.

You typically customize the commit operation to perform the following actions:

♦ Store the start and end time in a custom type inside the task data provider. Whenever you store the start and end time with custom types, you must make sure that the task chart handles the types you use or provide your own functions specified with the `taskStartTimeFunction` and `taskEndTimeFunction` properties.

♦ Store custom properties, other than the start time and end time that have been modified during editing.

For more information on using the `commitItemFunction` see *Updating the data provider item using a custom date format*.

Accessing task data and the task item renderer in an event listener

From within a Gantt sheet event listener you have access to the task data provider item, the current `startTime` and `endTime` of the task, and the item renderer used to display the task.

To access the data provider item you use the `item` property of the event.

To access the item renderer you use the `data` property of the `itemRenderer` of the event.

To access the current `startTime` and `endTime`, or the resource associated with the task, you use the `itemRenderer` property of the event.

For more details on the task item renderer and how it accesses data see *Default task item renderer in task chart*.

The following example shows an event listener for the `itemEditMove` event that accesses the data provider item, the task item renderer and the modified start and end time of the task.

```
private function onItemEditMove(event:GanttSheetEvent):void
{
   //...

   // The renderer
   var renderer:IDataRenderer = event.itemRenderer;

   // The TaskItem that holds the modified values of the task
   var taskItem:TaskItem = event.itemRenderer.data as TaskItem;

   // Modified values of the task
   var startTime:Date = taskItem.startTime;
   var endTime:Date = taskItem.endTime;

   // The data provider item
   var item:Object = event.item;

   // ...
}
```

Determining the kind of edit in an event listener

When the user starts editing a task, the Gantt sheet component determines the kind of edit operation from the item area where the user clicked. In the body of the event listeners for the `itemEditBegin`, `itemEditMove`, `itemEditResize`, and `itemEditEnd` events, you can determine the kind of edit and then handle it accordingly.

The `GanttSheet` class defines the `editKind` property, which contains a value that indicates the type of edit operation in progress. The `editKind` property has the following values defined in the class `TaskItemEditKind`.

Values of the kind of edit

Value	Description
MOVE_REASSIGN	Specifies that the user has clicked in the body of the task and is moving the task. You may get this value for `itemEditBegin`, `itemEditMove`, and `itemEditEnd` events.
RESIZE_END	Specifies that the user has clicked the end extremity of the task and is resizing it by changing the end time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.
RESIZE_START	Specifies that the user has clicked the start extremity of the task and is resizing it by changing the start time. You may get this value for `itemEditBegin`, `itemEditResize` and `itemEditEnd` events.

The following example shows how to retrieve the kind of edit within an event listener.

```
private function onItemEditBegin(event:GanttSheetEvent):void {
      var editKind:String = GanttSheet(event.target).editKind;
      //...
    }
```

Determining the reason for an itemEditEnd event

A user can end a task editing operation in several ways. In the body of the event listener for the `itemEditEnd` event, you can determine the reason for the event, and then handle it accordingly.

The `GanttSheetEvent` class defines the `reason` property, which contains a value that indicates the reason for the event. The `reason` property has the following values defined in the class `GanttSheetEventReason`.

Values of the reason property

Value	Description
CANCELLED	Specifies that the user cancelled editing and that they do not want to save the edited data.
COMPLETED	Specifies that the user completed the edit session, usually by releasing the mouse button.
OTHER	Specifies that the Gantt sheet is somehow in a state where editing is not allowed.

Examples of editing event handlers

Enforcing a minimum duration when resizing tasks

The following example in MXML shows how to use the `itemEditResize` event to enforce a minimum duration when resizing tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.HierarchicalData;
      import mx.events.FlexEvent;

      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import ilog.gantt.TaskItemEditKind;
      import ilog.gantt.TimeUtil;
      import ilog.utils.GregorianCalendar;
      import ilog.utils.TimeUnit;

      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0" },
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0" }
      ]);

      private function onItemEditResize(event:GanttSheetEvent):void
      {
        // Prevent the default behavior.
        event.preventDefault();

        var ganttSheet:GanttSheet = event.target as GanttSheet;
        var calendar:GregorianCalendar = ganttSheet.calendar;

        var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
        var startTime:Date = taskItem.startTime;
        var endTime:Date = taskItem.endTime;
        var duration:Number;

        if (ganttSheet.editKind == TaskItemEditKind.RESIZE_START) {
          // Snaps the edited start time on the
          // snapping time precision.
          startTime = calendar.round(startTime,
                                     ganttSheet.snappingTimePrecision.unit,
                                     ganttSheet.snappingTimePrecision.steps);


          // Ensure that the duration is at least 1 day.
          duration = endTime.time - startTime.time;
          if (duration < TimeUnit.DAY.milliseconds) {
            startTime.time = endTime.time - TimeUnit.DAY.milliseconds;
          }
          taskItem.startTime = startTime;
        }

        if (ganttSheet.editKind == TaskItemEditKind.RESIZE_END) {
          // Snaps the edited end time on the
          // snapping time precision.
          endTime = calendar.round(endTime,
                                   ganttSheet.snappingTimePrecision.unit,
                                   ganttSheet.snappingTimePrecision.steps);

          // Ensure that the duration is at least 1 day.
          duration = endTime.time - startTime.time;
          if (duration < TimeUnit.DAY.milliseconds) {
            endTime.time = startTime.time + TimeUnit.DAY.milliseconds;
          }
          taskItem.endTime = endTime;
        }
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                    taskDataProvider="{tasks}">
    <ilog:ganttSheet>
      <ilog:GanttSheet itemEditResize="onItemEditResize(event)" />
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example the listener for the `itemEditResize` event enforces the minimum duration of 1 day for tasks. It also mimics the behavior of the default listener by snapping the edited start time and end time using the `snappingTimePrecision` property of the `GanttSheet` object.

Custom snapping when moving tasks

The following example shows how to use the `itemEditMove` event to snap the start of a task when moving it.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.collections.HierarchicalData;
      import ilog.gantt.GanttSheet;
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import ilog.utils.TimeUnit;
      [Bindable]
      public var tasks:HierarchicalData = new HierarchicalData([
        { id: "T1", name: "Task #1",
          startTime: "1/14/2008 8:0:0", endTime: "1/28/2008 17:0:0"},
        { id: "T2", name: "Task #2",
          startTime: "1/29/2008 8:0:0", endTime: "2/5/2008 17:0:0"}
      ]);

      [Bindable]
      public var constraints:ArrayCollection = new ArrayCollection([
        { fromId:"T1", toId:"T2", kind:"endToStart" }
      ]);

      private function onItemEditMove(event:GanttSheetEvent):void
      {
        // Prevent the default behavior.
        event.preventDefault();

        // Snaps the start time on days boundary, starting at 08:00
        var taskItem:TaskItem = event.itemRenderer.data as TaskItem;
        var ganttSheet:GanttSheet = event.target as GanttSheet;
        var startTime:Date = ganttSheet.calendar.round(
                                                  taskItem.startTime,
                                                  TimeUnit.DAY, 1);
        startTime.hours = 8;

        // Move the item while preserving a constant duration.
        if (startTime.time != taskItem.startTime.time)
        {
          var duration:Number = taskItem.endTime.time
                              - taskItem.startTime.time;
          taskItem.startTime.time = startTime.time;
          taskItem.endTime.time = startTime.time + duration;
        }
      }
    ]]>
  </mx:Script>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                  taskDataProvider="{tasks}"
                  constraintDataProvider="{constraints}">
    <ilog:ganttSheet>
      <ilog:GanttSheet itemEditMove="onItemEditMove(event)" />
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example the listener for the `itemEditMove` event snaps the start time of tasks to 08:00 for each day.

Updating the data provider item using a custom date format

The following example shows how to access dates stored with a custom date format. The start and end time are read from the data provider item using custom functions that parse the custom date format. A custom function is used to update the data provider item after editing is complete using the custom date format for the start time and end time.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import ilog.core.DataItem;
      import ilog.gantt.GanttSheetEventReason;
      import ilog.gantt.GanttSheet;
      import ilog.gantt.GanttSheetEvent;
      import ilog.gantt.TaskItem;
      import mx.formatters.DateFormatter;

      private var replaceDashes:RegExp = new RegExp("-","g");
      private var replaceT:RegExp = new RegExp("T");
      private var iso8601Formatter:DateFormatter = new DateFormatter();

      private function initApp():void {
        iso8601Formatter.formatString = "YYYY-MM-DDTJJ:NN:SS";
      }

      private function readStartDate(item:Object):Date {
        return readDate(item, "startTime");
      }

      private function readEndDate(item:Object):Date {
        return readDate(item, "endTime");
      }

      private function readDate(item:Object, dateField:String):Date {
        var dateString:String = item[dateField].replace(replaceDashes, "/");
        dateString = dateString.replace(replaceT, " ");
        return new Date(dateString);
      }

      // When setting taskStartTimeFunction or taskEndTimeFunction
      // you must provide a commitItemFunction to update the data
      // provider item. In this case we update the start and end
      // time in the data provider item using a ISO 8601 date
      // formatter.
      private function commitItem(dataItem:DataItem):void {
        if (dataItem is TaskItem)
        {
          var item:Object = dataItem.data;
          var taskItem:TaskItem = dataItem as TaskItem;

          item["startTime"] = iso8601Formatter.format(taskItem.startTime);
          item["endTime"] = iso8601Formatter.format(taskItem.endTime);
        }
      }
    ]]>
  </mx:Script>

  <!-- The dates in the model are expressed using the ISO 8601 Extended
       Format for date and time of day representation. For more details
       on this format see:
       "http://isotc.iso.org/livelink/livelink/4021199/ISO_8601_2004_E.zip?
       func=doc.Fetch&nodeid=4021199"
   -->
  <mx:Model id="model">
    <model>
      <task id="T1" name="Define project vision"
            startTime="2008-01-14T08:00:00" endTime="2008-01-15T17:00:00"/>
      <task id="T2" name="Refine project vision"
            startTime="2008-01-16T08:00:00" endTime="2008-01-17T17:00:00"/>
      <task id="T3" name="Assess project vision"
            startTime="2008-01-18T18:00:00" endTime="2008-01-21T17:00:00"/>
    </model>
  </mx:Model>

  <ilog:TaskChart id="taskChart" width="100%" height="100%"
                      taskDataProvider="{model.task}"
                      taskStartTimeFunction="readStartDate"
                      taskEndTimeFunction="readEndDate"
                      >
    <ilog:ganttSheet>
      <ilog:GanttSheet commitItemFunction="commitItem">
      </ilog:GanttSheet>
    </ilog:ganttSheet>
  </ilog:TaskChart>
</mx:Application>
```

In this example the start and end time of tasks are stored in the task data provider using the ISO8601:2000 extended format. Custom functions are set on the `taskStartTimeFunction` and `taskEndTimeFunction` properties of the task chart control. These functions parse the ISO8601:2000 formatted String and retrieve Date objects, stored in the `TaskItem` object associated with the task data provider item. A custom function is set on the `commitItemFunction` property of the Gantt sheet. It uses a custom date formatter to store the start and end time as Strings in the ISO8601:2000 format.

Working with Date objects

IBM ILOG Elixir provides a set of utility classes to help manipulate `Date` objects:

- ♦ `ilog.utils.TimeUnit`, defines time units
- ♦ `ilog.utils.GregorianCalendar`, provides operations for accurately manipulating Date objects

The task chart control uses an instance of `GregorianCalendar` that you can retrieve from the `ResourceChart.calendar` property. It is important to use the same instance of `GregorianCalendar` that is used by the task chart control to achieve consistent results.

The `GregorianCalendar` class provides the following methods:

- ♦ `addUnits`, allows the adding of time units to a `Date` object or the removing of time units from a `Date` object.
- ♦ `floor`, gets the floor of a `Date` object for a given time unit.
- ♦ `getWeek`, returns the number of the week for a `Date` object.
- ♦ `round`, rounds a `Date` object to the nearest time unit, for a given time unit.

For complete reference information, see the `TimeUnit` class and the `GregorianCalendar` class.

For more information on `Date`, see *Working with dates and times* in *Adobe Programming ActionScript 3.0*.

Gauges and indicators

Describes the use of IBM ILOG Elixir gauges and indicators with Adobe® Flex® 3.

In this section

Gauges
Describes the use of IBM ILOG Elixir gauges with Adobe® Flex® 3.

Indicators
Describes the use of IBM ILOG Elixir indicators with Adobe® Flex® 3.

Gauges

Describes the use of IBM ILOG Elixir gauges with Adobe® Flex® 3.

In this section

Introduction to gauges
Discusses gauge components and describes the two kinds of gauges, predefined and custom, and their uses.

Gauges architecture
Describes the architecture of the gauge classes.

Using and customizing predefined gauges
Describes how to use and customize predefined circular gauges.

Building custom gauges
Describes how to create custom circular and custom rectangular gauges.

Introduction to gauges

Gauge components

IBM ILOG Elixir Gauges help improve data visualization applications by using gauges in the context of analysis and real-time monitoring of data. IBM ILOG Elixir Gauges include predefined circular and rectangular (horizontal and vertical) gauges, knobs, and dials, as well as a dedicated framework to construct custom gauges.

Gauge components are interactive. You can edit the data values displayed by the gauges using the mouse. Gauges also provide customizable animations on data changes.

IBM ILOG Elixir provides *predefined* gauges and a framework for building *custom* gauges.

If you need to display a single value on a single scale and if the layouts of the predefined gauges satisfy your requirements, use or customize the predefined gauges. See *Using and customizing predefined gauges*.

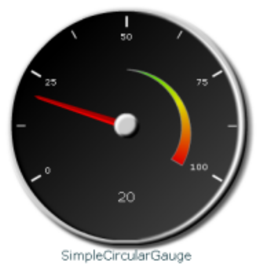
If you need to display several values or scales, or need to apply more advanced layouts to the gauge elements, use the Gauges framework to build your own custom gauges. See *Building custom gauges*.

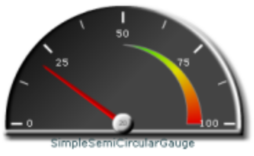
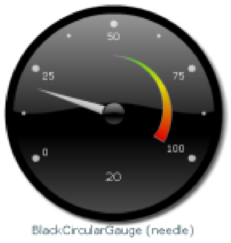
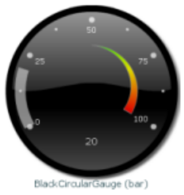
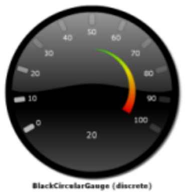
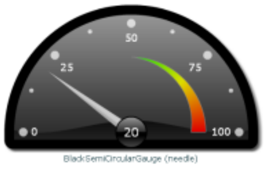
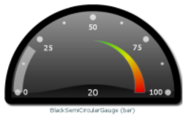
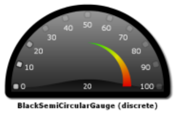
Predefined gauges

Predefined gauges are preconfigured Flex® components which expose a simple interface to the developer. They expose a single value on a single scale with a predefined layout of the elements of the gauge. For example, the predefined circular gauge displays a single value on a circular scale and is configured by default with a single needle.

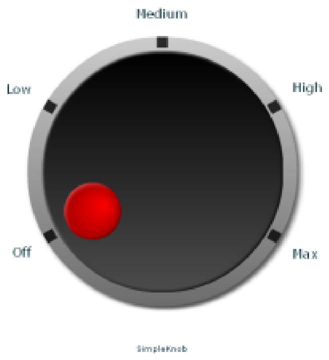
The following tables give an overview of the predefined gauges: circular gauges, knobs, horizontal gauges, and vertical gauges.

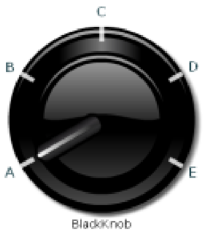
Circular gauges

SimpleCircularGauge			

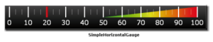
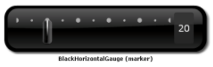
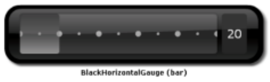
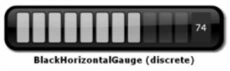
SimpleSemiCircularGauge			
BlackCircularGauge	 0 25 50 75 100 20 BlackCircularGauge (needle)	 0 25 50 75 100 20 BlackCircularGauge (bar)	 0 10 20 30 40 50 60 70 80 90 100 20 BlackCircularGauge (discrete)
BlackSemiCircularGauge	 0 25 50 75 100 20 BlackSemiCircularGauge (needle)	 0 25 50 75 100 20 BlackSemiCircularGauge (bar)	 0 10 20 30 40 50 60 70 80 90 100 20 BlackSemiCircularGauge (discrete)

Knobs

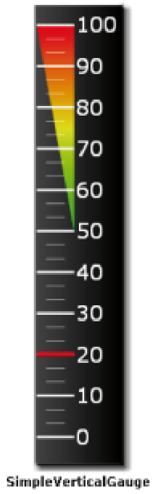
SimpleKnob	

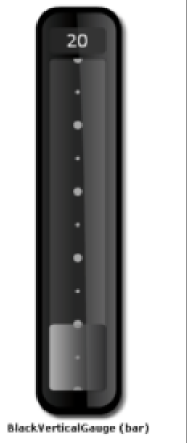
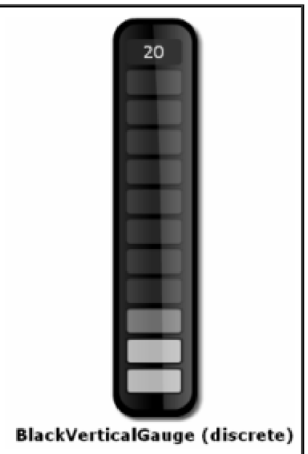
BlackKnob	

Horizontal gauges

SimpleHorizontalGauge			
BlackHorizontalGauge			

Vertical gauges

SimpleVerticalGauge	 100 90 80 70 60 50 40 30 20 10 0 SimpleVerticalGauge		

BlackVerticalGauge			

Custom gauges

IBM ILOG Elixir gauges rely on a generic gauges framework which allows you to easily build custom MXML or ActionScript® components representing gauges. For example, a custom circular gauge can handle any number of values displayed by several needles.

Predefined gauges may not satisfy all the use cases. That is why, for specific needs, the IBM ILOG Elixir framework allows you to make custom gauges with an arbitrary number of elements. Gauges defined with the framework are composite components with interaction and animation. With this approach, there is no limitation. A gauge is a set of elements with layout and configuration properties. The geometry of all elements, such as scales or needles, is controlled by properties which, most of the time, can be expressed as a percentage so that the gauge scales correctly when resized.

Gauges architecture

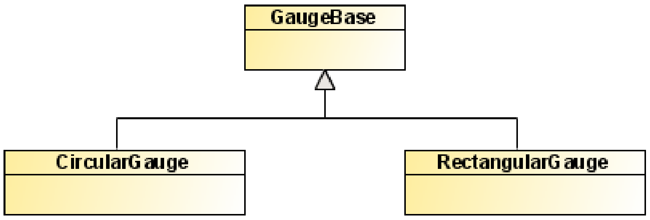
The architecture of the gauge components can be broken down into three categories of classes: the classes of the framework, the classes of the predefined circular gauges, and the classes of the predefined rectangular gauges. The following UML diagrams show the relationships between the classes within each category.

Classes of the gauges framework

Gauges framework classes are composed of: base classes, gauge scales classes, and visual elements classes for circular and rectangular gauges.

Base classes

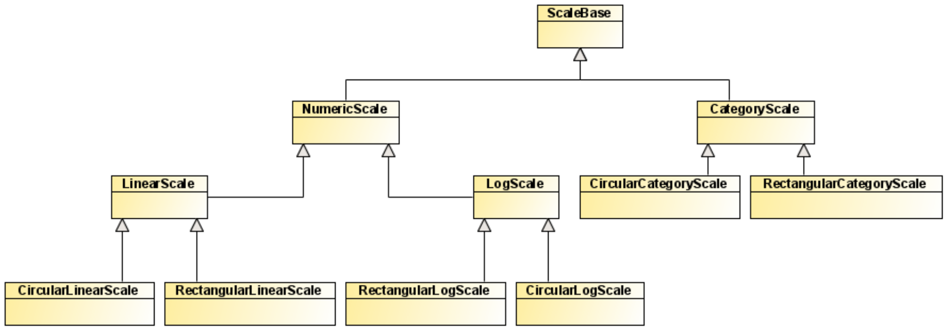
The framework is designed so that a gauge is defined by a set of logical scales and a set of visual elements. These properties are defined on the `GaugeBase` class.



The `CircularGauge` and `RectangularGauge` subclasses manage the specificities of each type of gauge, including the layout of the various visual elements.

Gauge scale classes

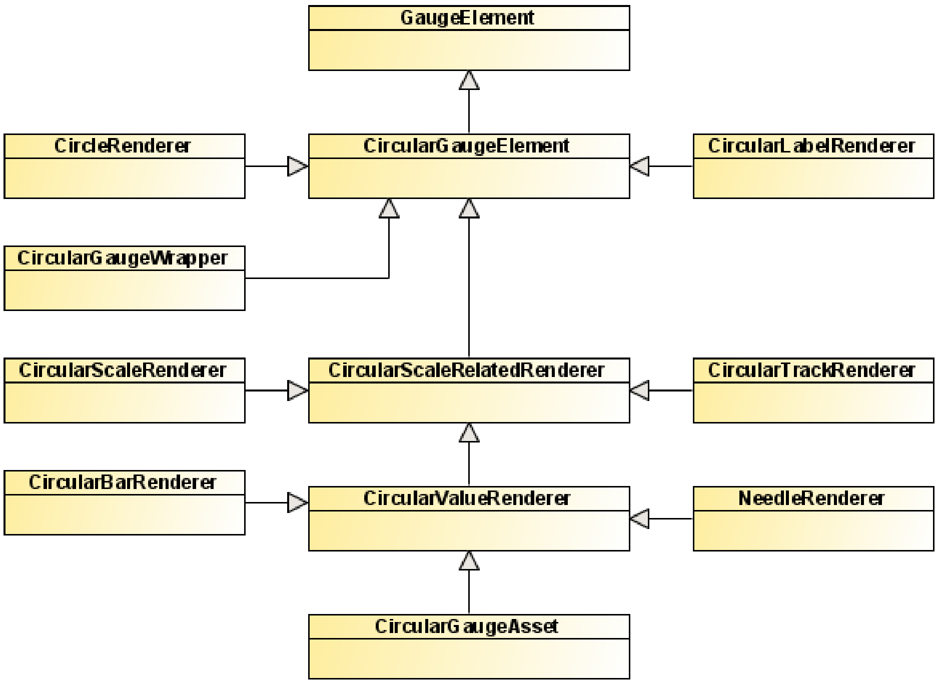
The following diagram shows the logical scales hierarchy of the framework. There are base classes common to circular and rectangular gauges, as well as specific classes.



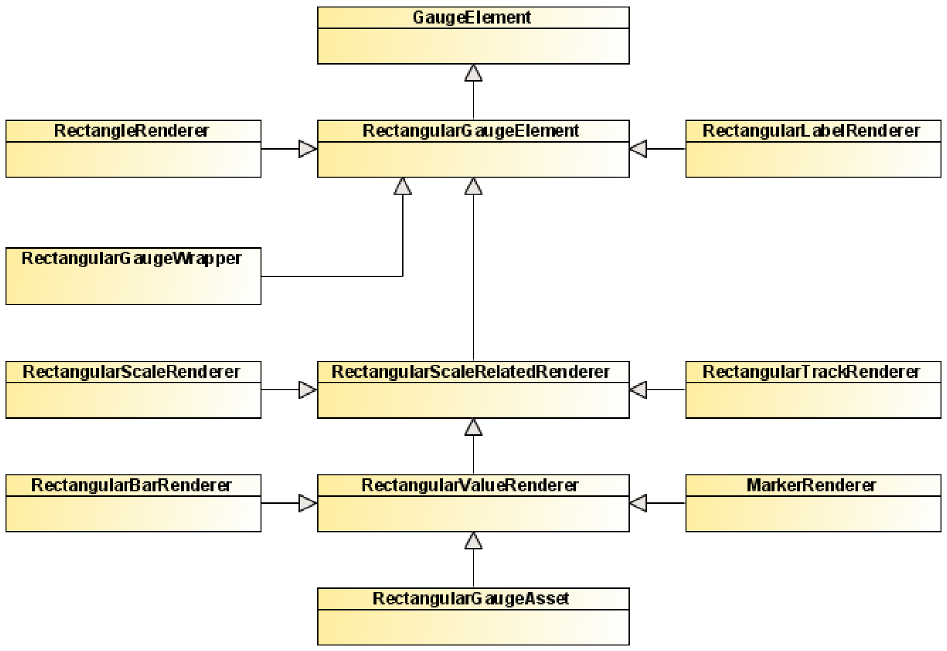
A scale defines the logical model of a gauge. It defines which values are acceptable for the scale: either from a minimum to a maximum for `NumericScale` subclasses, or from a set of values for `CategoryScale` subclasses. Scales are used by some visual elements to make sure they render correctly according to their scale.

Visual elements classes for circular and rectangular gauges

The following diagram shows the hierarchy of the visual elements for circular gauges.



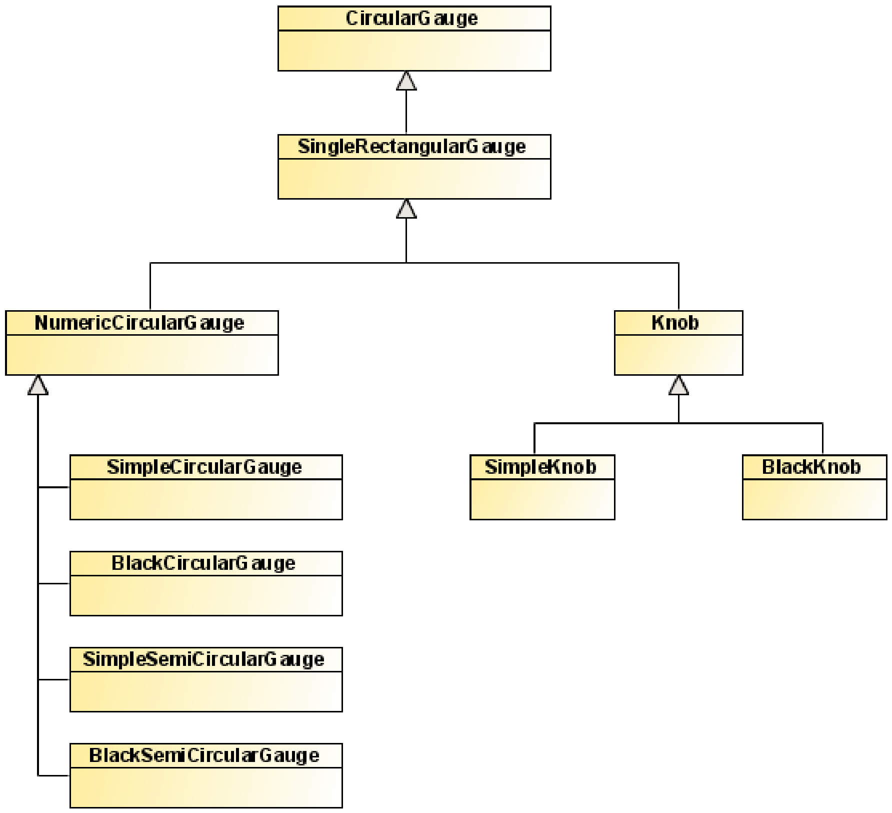
The following diagram shows the hierarchy of the visual elements for rectangular gauges.



All elements inheriting from `CircularScaleRelatedRenderer` or `RectangularScaleRelatedRenderer` need a scale to render correctly. If no scale is set on such a visual element, the first element of the scale array of the gauge is used. Elements that relate to the scale are elements that need to render a scale (for example, ticks) or something that represents the scale value. Other types of elements do not rely on the scale because they do not need the scale (for example, decorations, labels).

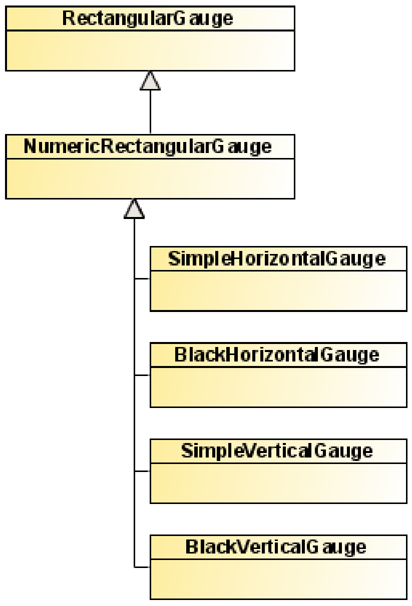
Classes of the predefined circular gauges

The following diagram shows the predefined circular gauges classes.



Classes of the predefined rectangular gauges

The following diagram shows the predefined circular gauges classes.


RectangularGauge
NumericRectangularGauge
SimpleHorizontalGauge
BlackHorizontalGauge
SimpleVerticalGauge
BlackVerticalGauge

Using and customizing predefined gauges

Describes how to use and customize predefined circular gauges.

In this section

About predefined gauges
Describes the two types of predefined gauges.

Predefined circular gauges
Describes how to use predefined circular gauges with an example.

Customizing predefined circular gauges
Describes how to customize predefined circular gauges with an example of the code.

Predefined rectangular gauges
Describes how to use predefined rectangular gauges with an example.

Customizing predefined rectangular gauges
Describes how to customize predefined rectangular gauges with an example of the code.

Predefined knobs
Describes how to use predefined predefined knobs with an example.

Customizing predefined knobs
Describes how to customize predefined knobs with an example of the code.

About predefined gauges

Predefined gauges are MXML components built on top of the gauges framework. They represent a single value on a single scale with a predefined layout of the visual elements.

Predefined gauges are designed to clearly expose important properties and hide implementation details like the geometry of gauge elements.

IBM ILOG Elixir provides two types of predefined gauges:

♦ Fully programmatic gauges

  These gauges can be found in the package `ilog.gauges.controls.simple`

♦ Gauges based on external assets, typically built with Flash™ CS3 or Illustrator CS3 to produce embeddable vector graphics.

  These gauges can be found in the package `ilog.gauges.controls.black`

You can use predefined gauges as-is or customize them in many ways. For example, you can change their colors and customize or replace the default scale, default track, and default value renderer.

Predefined circular gauges

Full circular gauges

Full circular gauges have a rotation point at the center of the gauge and an aspect ratio of 1.0. IBM ILOG Elixir provides two full circular gauges:

♦ `BlackCircularGauge` (black-style skinned gauge)

♦ `SimpleCircularGauge` (full programmatic gauge)

The following code shows how to use some properties, including style properties, of predefined circular gauges:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:BlackCircularGauge width="100%" height="100%"
     value="20" minimum="0" maximum="100"
     orientation="clockwise" startAngle="210" endAngle="330"
     majorTickInterval="10" minorTickInterval="1"
     skinColor="0xaa1111" />
  <ilog:SimpleCircularGauge width="100%" height="100%"
     value="20" minimum="0" maximum="100"
     orientation="cclockwise" startAngle="0" endAngle="180"
     majorTickInterval="20" minorTickInterval="10"
     backgroundColors="[0xaa1111, 0x111111]"/>
</mx:Application>
```

Semicircular gauges

Semicircular gauges have a scale going from 0 to 180 degrees (or conversely, depending on their orientation). They have an aspect ratio greater than 1.0 and the elements have a y origin at a position greater than 50%. IBM ILOG Elixir provides two semicircular gauges:

♦ `BlackSemiCircularGauge` (black-style skinned gauge)

♦ `SimpleSemiCircularGauge` (full programmatic gauge)

Semicircular gauges can be configured in a similar way to full circular gauges.

Customizing predefined circular gauges

You can go one step further than just styling predefined gauges and customize them by replacing some of their inner elements. The following code shows how to:

- ♦ Replace the default scale by a logarithmic scale
- ♦ Customize track colors
- ♦ Replace the default value renderer (`NeedleRenderer`) with a `CircularBarRenderer`
- ♦ Make sure the value renderer is editable so that you can drag the bar to change the gauge value
- ♦ Use an indicator label function to specialize the display of the label

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
  backgroundColor="0xFFFFFF">
  <mx:Script>
  <![CDATA[
    public function labelFunction(value:Object):String {
      return String(Math.round(value as Number));
    }
  ]]>
  </mx:Script>
  <ilog:SimpleCircularGauge width="100%" height="100%"
    minimum="1"
    maximum="100000"
    trackMinimum="1000"
    trackMaximum="100000"
    value="10"
    editable="true"
    indicatorLabelFunction="labelFunction" >
    <ilog:scale>
      <ilog:CircularLogScale/>
    </ilog:scale>
    <ilog:track>
      <ilog:CircularTrackRenderer radius="30%">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0x005500" ratio="0"/>
          <mx:GradientEntry color="0x00AA00" ratio="0.2"/>
          <mx:GradientEntry color="0x00FF00" ratio="0.4"/>
          <mx:GradientEntry color="0xAA0000" ratio="0.8"/>
          <mx:GradientEntry color="0xFF0000" ratio="1.0"/>
        </ilog:gradientEntries>
      </ilog:CircularTrackRenderer>
    </ilog:track>
    <ilog:valueRenderer>
      <ilog:CircularBarRenderer radius="45%" startThickness="30%"
                                endThickness="30%">
        <ilog:gradientEntries>
```

```
          <mx:GradientEntry color="0x00FF00" ratio="0" alpha="0.1"/>
          <mx:GradientEntry color="0xFF0000" ratio="1" alpha="0.8"/>
        </ilog:gradientEntries>
      </ilog:CircularBarRenderer>
    </ilog:valueRenderer>
  </ilog:SimpleCircularGauge>
</mx:Application>
```

Predefined rectangular gauges

Horizontal gauges

Horizontal gauges are rectangular gauges displayed in the horizontal direction only (their `direction` attribute is set to `horizontal`). In IBM ILOG Elixir, the predefined horizontal gauges inherit from the `NumericRectangularGauge` base class which provides a simple API for classes that need to display a numeric value along a single numeric scale (`RectangularLinearScale` or `RectangularLogScale`).

IBM ILOG Elixir provides two horizontal gauges:

- ♦ `BlackHorizontalGauge` (black-style skinned gauge)
- ♦ `SimpleHorizontalGauge` (full programmatic gauge)

The following code shows how to use some properties, including style properties, of predefined horizontal gauges:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Style>
    .tickStyle {
      fill : blue;
    }
  </mx:Style>
  <ilog:BlackHorizontalGauge width="100%"
                             value="500" minimum="0" maximum="1000"
                             majorTickInterval="100"
                             minorTickInterval="50"
                             majorTickStyleName="tickStyle"
                             minorTickStyleName="tickStyle"
                             skinColor="0x11ff11" />
  <ilog:SimpleHorizontalGauge width="100%" title="The Title"
                              value="200" minimum="0" maximum="1000"
                              trackMinimum="250"
                              trackMaximum="1000"
                              majorTickInterval="200"
                              minorTickInterval="100"
                              backgroundColors="[0x11ff11, 0x111111]"/>
</mx:Application>
```

Vertical gauges

Vertical gauges are rectangular gauges displayed in the vertical direction only (their `direction` attribute is set to `vertical`). In IBM ILOG Elixir, the predefined vertical gauges inherit from the `NumericRectangularGauge` base class which provides a simple API for classes that need to display a numeric value along a single numeric scale (`RectangularLinearScale` or `RectangularLogScale`).

IBM ILOG Elixir provides two vertical gauges:

- ♦ `BlackVerticalGauge` (black-style skinned gauge)
- ♦ `SimpleVerticalGauge` (full programmatic gauge)

The following code shows how to use some properties, including style properties, of predefined vertical gauges:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Style>
    .myStyle (
      color : green;
    )
  </mx:Style>
  <ilog:BlackVerticalGauge height="100%"
                           value="500" minimum="0" maximum="1000"
                           majorTickInterval="100"
                           minorTickInterval="50"
                           skinColor="0x1111ff"
                           color="red" />
  <ilog:SimpleVerticalGauge height="100%" title="The Title"
                            value="200" minimum="0" maximum="1000"
                            trackMinimum="250" trackMaximum="1000"
                            majorTickInterval="200"
                            minorTickInterval="100"
                            backgroundColors="[0x1111ff, 0x111111]"
                            color="red"
                            valueIndicatorStyleName="myStyle" />
</mx:Application>
```

Customizing predefined rectangular gauges

You can go one step further than just styling predefined gauges and customize them by replacing some of their inner elements. The following code shows how to:

- ♦ Replace the default scale by a logarithmic scale
- ♦ Customize track colors
- ♦ Replace the default value renderer (`MarkerRenderer`) with a `RectangularBarRenderer`
- ♦ Make sure the value renderer is editable so that you can drag the bar to change the gauge value
- ♦ Use an indicator label function to specialize the display of the label

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      public function labelFunction(value:Object):String {
        return String(Math.round(value as Number));
      }
    ]]>
  </mx:Script>
  <ilog:SimpleHorizontalGauge width="100%" value="10" editable="true"
       minimum="1" maximum="100000"
       trackMinimum="1000" trackMaximum="100000"
       indicatorLabelFunction="{labelFunction}">
    <ilog:scale>
      <ilog:RectangularLogScale/>
    </ilog:scale>
    <ilog:track>
      <ilog:RectangularTrackRenderer>
        <ilog:gradientEntries>
          <mx:GradientEntry color="0x005500" ratio="0"/>
          <mx:GradientEntry color="0x00AA00" ratio="0.2"/>
          <mx:GradientEntry color="0x00FF00" ratio="0.4"/>
          <mx:GradientEntry color="0xAA0000" ratio="0.8"/>
          <mx:GradientEntry color="0xFF0000" ratio="1.0"/>
        </ilog:gradientEntries>
      </ilog:RectangularTrackRenderer>
    </ilog:track>
    <ilog:valueRenderer>
      <ilog:RectangularBarRenderer>
        <ilog:fill>
          <mx:LinearGradient>
            <mx:GradientEntry color="0x00FF00" ratio="0" alpha="0.1"/>
            <mx:GradientEntry color="0xFF0000" ratio="1" alpha="0.8"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:RectangularBarRenderer>
    </ilog:valueRenderer>
  </ilog:SimpleHorizontalGauge>
</mx:Application>
```

Predefined knobs

IBM ILOG Elixir provides two predefined knobs:

- `BlackKnob` (black-style skinned knob)
- `SimpleKnob` (full programmatic knob)

The following code shows how to use some properties of predefined knobs.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
  layout="horizontal"
  backgroundColor="0xFFFFFF">
  <ilog:BlackKnob width="100%" height="100%"
    categories="['OFF', 'ON']"/>
  <ilog:SimpleKnob width="100%" height="100%"
    value="0" categories="['0', '1', '2', '3', '4', 'MAX']"/>
</mx:Application>
```

Customizing predefined knobs

The following code shows how to:

- ♦ Change the orientation of the black knob
- ♦ Customize start and end angles
- ♦ Change background colors of the simple knob

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
  layout="horizontal"
  backgroundColor="0xFFFFFF">
  <ilog:BlackKnob width="100%" height="100%"
    value="ON"
    orientation="cclockwise"
    startAngle="140"
    endAngle="40"
    categories="['OFF', 'ON']"/>
  <ilog:SimpleKnob width="100%" height="100%"
    value="20"
    orientation="clockwise"
    startAngle="210"
    endAngle="330"
    categories="['0', '1', '2', '3', '4', 'MAX']"
    backgroundColors="[0xEEEEEE, 0xCCCCCC, 0xEEEEEE]"/>
</mx:Application>
```

Building custom gauges

Describes how to create custom circular and custom rectangular gauges.

In this section

About the gauges framework
Discusses the layer that underlies circular and rectangular gauges.

Custom circular gauges
Describes the elements that make up a circular gauge and how to configure them to create a circular gauge from scratch.

Custom rectangular gauges
Describes the elements that make up a rectangular gauge and how to configure them to create a rectangular gauge from scratch.

About the gauges framework

IBM ILOG Elixir gauges are based on a low-level framework that allows you to build:

- Circular gauges — Any gauge on which the value is indicated by a rotating element
- Rectangular gauges — Any vertical or horizontal gauge

The framework contains classes that are common to both circular and rectangular gauges, for example, the low-level scale-related classes. The other classes are specific to either circular or rectangular gauges.

The framework provides default implementations for most of the gauges elements, like scale renderers, needles, markers, or tracks.

Custom circular gauges

Describes the elements that make up a circular gauge and how to configure them to create a circular gauge from scratch.

In this section

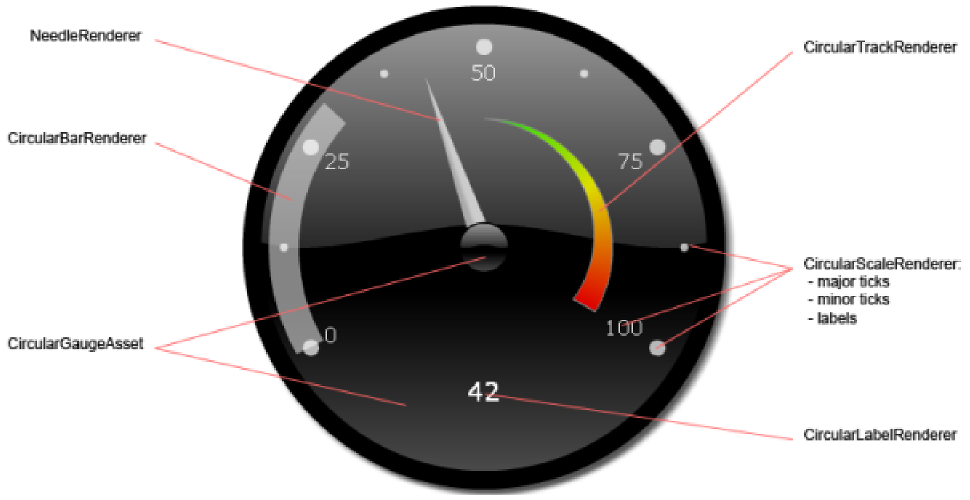
Overview of a circular gauge
Illustrates the elements that compose a circular gauge.

The logical scale(s) of circular gauges
The first thing to do when creating a custom gauge is to define the scale(s) it will be using.

The visual elements of circular gauges
Describes the main visual elements of circular gauges and how to customize them. Information on `CircularGaugeWrapper`is in the reference manual only.

Building a circular gauge step by step
Explains how to build a circular gauge from scratch.

Overview of a circular gauge



The visual elements of a circular gauge extend the `CircularGaugeElement` class or one of its subclasses. They are characterized by:

- A reference length to compute the position and size of the element. The length is either the width or the height of the element depending on the value of the `radiusRelativeTo` property.
- An origin point defined by the `originX` and `originY` properties. It can be expressed in a percentage of the reference length.
- A radius defined by the `radius` property. It is usually expressed in a percentage of the reference length.

The layout of the visual elements of a circular gauge is processed in the following order:

1. The drawing area bounds of the gauge are computed according to padding values, to the space needed for the optional title to be displayed, and to the `aspectRatio` value

   The `aspectRatio` property defines the width/height ratio. By default, it is set to 1.0, which means that the drawing area is a centered square as large as possible once the padding and the title space has been taken into account.

2. Each element of the elements array is positioned and sized according to their `originX`, `originY`, `radius`, and `radiusRelativeTo` properties.

The logical scale(s) of circular gauges

A scale is a logical element that handles the main characteristics of a gauge: minimum and maximum values (numeric scales), categories (category scales), start and end angles, orientation. A scale can be numeric or category-based. Scales are used by many visual elements to render themselves. For example, the `CircularScaleRenderer` or the `NeedleRenderer` use their associated scale to render themselves.

Available circular scales are:

- `CircularLinearScale`
- `CircularLogScale`
- `CircularCategoryScale`

The following example defines a logical scale with its associated scale renderer and a needle pointing a specific value.

```
<ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale
        minimum="10" maximum="100"
        majorTickInterval="10" minorTickInterval="5"
        startAngle="0" endAngle="180"
        orientation="cclockwise"/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer/>
      <ilog:NeedleRenderer value="30" radius="30%"/>
    </ilog:elements>
 </ilog:CircularGauge>
```

If the scale attribute is not specified on a visual element, the visual element is automatically associated with the first logical scale in the scales array. If you need to reference a different scale, use the `scale` property.

A scale provides two arrays of `TickItem`, one for major ticks and one for minor ticks, that are used by the `CircularScaleRenderer` to position the ticks. By default, the arrays are automatically computed, but you can override the default behavior by setting the scale properties, for example, the desired interval between two ticks.

Note: CircularCategoryScale and CircularLogScale do not generate minor tick items.

The visual elements of circular gauges

Describes the main visual elements of circular gauges and how to customize them. Information on `CircularGaugeWrapper`is in the reference manual only.

In this section

The CircularScaleRenderer class
Describes the `CircularScaleRenderer`class and gives code examples.

About value renderer classes
Introduces the value renderer classes: `NeedleRenderer`, `CircularBarRenderer`, and `CircularGaugeAsset`.

The NeedleRenderer class
Describes the `NeedleRenderer` class and gives code examples.

The CircularBarRenderer class
Describes the `CircularBarRenderer` class and gives code examples.

The CircularTrackRenderer class
Describes the `CircularTrackRenderer` class and gives code examples.

The CircularLabelRenderer class
Describes the `CircularLabelRenderer` class and gives code examples.

The CircleRenderer class
Describes the `CircleRenderer` class and gives code examples.

The CircularGaugeAsset class

Describes the `CircularGaugeAsset` class and gives code examples.

The CircularScaleRenderer class

The scale renderer is the visual element responsible for rendering the ticks and labels of a scale. The scale renderer class uses data provided by the logical scale, that is:

- `majorTicks` and `minorTicks` arrays
- `startAngle`, `endAngle`, `orientation`

Formatting tick marks

You can customize the rendering of the `CircularScaleRenderer` ticks by setting the `majorTickRenderer` property for major ticks and the `minorTickRenderer` property for minor ticks. If the specified renderer implements `IDataRenderer`, the associated `TickItem` is set to the data property of the renderer.

The gauges framework includes two default tick item renderers: `RectangleTickRenderer` and `CircleTickRenderer`. The size of the tick renderer instances is controlled by the properties `majorTickWidth`, `majorTickLength`, `minorTickWidth`, and `minorTickLength`. These properties can all be expressed in pixels or in a percentage of the `radius` property.

The placement of the tick renderer instances is based on the value of the `tickPlacement` property. Possible values are: `inside`, `outside`, or `cross`.

The following example shows how to format the tick marks of a `CircularScaleRenderer` and change the renderer for major ticks:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:CircularGauge>
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer
        tickPlacement="cross"
        majorTickWidth="10%" majorTickLength="10%"
        minorTickWidth="2%" minorTickLength="5%">
        <ilog:majorTickRenderer>
          <mx:Component>
            <ilog:CircleTickRenderer>
              <ilog:fill>
                <mx:SolidColor color="0xFF0000"/>
              </ilog:fill>
            </ilog:CircleTickRenderer>
          </mx:Component>
        </ilog:majorTickRenderer>
      </ilog:CircularScaleRenderer>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Formatting labels

You can size and format the scale labels using the properties `labelRadius`, `labelPlacement`, and `labelFunction`.

The following example shows how to place a label outside the tick marks with a customized text and a relative size of 10%:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.gauges.TickItem;
      private function withDegrees(t:TickItem):String {
        return t.value + "°";
      }
    ]]>
  </mx:Script>
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer
        radius="30%"
        labelRadius="100%"
        labelFontSize="10%"
        labelPlacement="outside"
        labelFunction="{withDegrees}"
        />
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Defining a custom renderer

You can also define a custom renderer for rendering labels of the scale. The following example shows how to use a `NumericIndicator` to render the labels.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0x000000">
  <ilog:CircularGauge>
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer>
        <ilog:labelRenderer>
          <mx:Component>
            <ilog:NumericIndicator numChars="0"/>
          </mx:Component>
        </ilog:labelRenderer>
      </ilog:CircularScaleRenderer>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Note that if you redefine the renderer, the `labelFontSize` property is ignored.

About value renderer classes

A value renderer is a renderer class that represents a data value. The circular gauges framework provides three predefined value renderers:

- `NeedleRenderer` — A customizable needle that points the value of the renderer. See *The NeedleRenderer class* for details.
- `CircularBarRenderer` — A customizable bar that starts from the start angle of the scale and extends to the angle corresponding to the value of the renderer. See *The CircularBarRenderer class* for details.
- `CircularGaugeAsset` — A value renderer that delegates the rendering to an `IFlexDisplayObject`, which usually represents an external asset. See *The CircularGaugeAsset class* for details.

These renderers extend the `CircularValueRenderer` class, which can be customized using the following properties:

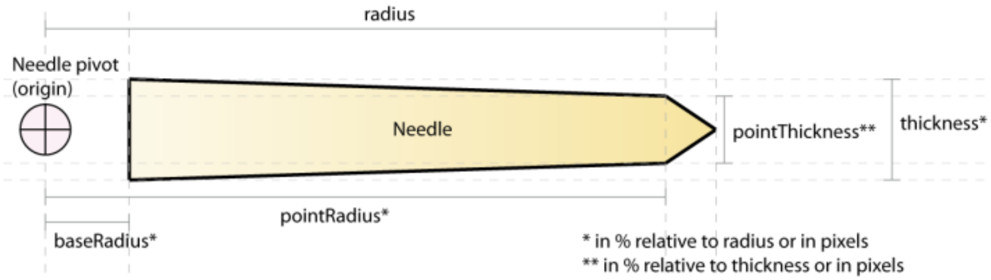
- `value` — The value the renderer represents.
- `editable` — This property indicates whether the value can be edited using mouse gestures.
- `liveDragging` — If this property is set to true, the value of the renderer changes during user interaction and not only after the interaction (which is the default behavior).
- `editMode` — The possible values of this property are: `discrete` and `continuous`. If the property is set to `discrete`, the value renderer snaps to the scale accepted values (categories for `CircularCategoryScale`, or value on the snapping interval for `CircularLinearScale`) when the user moves the mouse. Otherwise, the rendered value moves continuously.
- `mouseMode` — This property defines the reactive area of the renderer during an interaction. If it is set to `shape`, only the drawn area of the renderer reacts to the mouse. Otherwise, the reactive area corresponds to a circle defined by the `originX`, `originY`, and `radius` of the renderer.

The NeedleRenderer class

The needle renderer is a visual element responsible for rendering the needle of the gauge. It references a logical scale to point to the angle defined by its `value` property. A needle can also be represented by an external asset, see *The CircularGaugeAsset class*.

Customizing the geometry of a NeedleRenderer

The following illustration shows the main properties of a needle to help you customize its geometry.



The following code example shows four options to customize the needle geometry:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF"
                >
  <mx:Script>
    <![CDATA[
      import mx.graphics.SolidColor;
    ]]>
  </mx:Script>
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer/>
      <ilog:NeedleRenderer
        radius="40%"
        baseRadius="0%"
        thickness="5%"
        pointThickness="100%"
        pointRadius="95%"
        value="0"
        editable="true"
        fill="{new SolidColor(0xAA0000)}" />
      <ilog:NeedleRenderer
        radius="40%"
        baseRadius="80%"
        thickness="10%"
        pointThickness="100%"
        pointRadius="95%"
        value="20"
        editable="true"
        fill="{new SolidColor(0xAAAA00)}" />
      <ilog:NeedleRenderer
        radius="40%"
        baseRadius="0%"
        thickness="10%"
        pointThickness="50%"
        pointRadius="95%"
        value="50"
        editable="true"
        fill="{new SolidColor(0xAA00AA)}" />
      <ilog:NeedleRenderer
        radius="40%"
        baseRadius="0%"
        thickness="10%"
        pointThickness="0%"
        pointRadius="100%"
        value="80"
        editable="true"
        fill="{new SolidColor(0x00AAAA)}" />
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

The fill and stroke of a needle can also be changed with the `fill` and `stroke` style properties.

The CircularBarRenderer class

This class represents an annulus with a start angle equal to the start angle of the scale and an end angle that corresponds to the value of the renderer.

Customizing the geometry of a CircularBarRenderer
The geometry of a `CircularBarRenderer` is controlled by the following properties:

♦ `startThickness` — The thickness of the bar at the start angle of the bar (which is also the start angle of the scale).

♦ `endThickness` — The thickness of the bar at the end angle of the scale. The thickness at the end angle of the bar that corresponds to the value of the renderer is automatically computed from that.

♦ `placement` — The position of the annulus relative to the radius. Possible values are: `inside` (default), `outside`, and `cross`.

`startThickness` and `endThickness` can be expressed in a percentage of the radius or in pixels. The default value is 20% of the radius.

Customizing the colors of a CircularBarRenderer
The `CircularBarRenderer` supports two fill modes:

♦ Solid color — This is the default mode. The style properties `rendererColor` and `rendererAlpha` are used to fill the bar.

♦ Color gradients — Fill the `gradientEntries` array with `GradientEntry` instances to obtain a conical gradient from the scale's start angle to the scale's end angle. This gradient is cut at the bar renderer current value.

The following code example shows different scenarios of bar renderers:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.graphics.*;
    ]]>
  </mx:Script>
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer/>
      <ilog:CircularBarRenderer
        radius="16%"
        value="100"
        editable="true"
        rendererColor="0x0000FF"
        stroke="{new Stroke(0xAAAAAA, 5)}" />
      <ilog:CircularBarRenderer
```

```
        radius="26%"
        value="80"
        placement="outside"
        startThickness="0"
        editable="true">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0xFF0000" ratio="0"/>
          <mx:GradientEntry color="0x00FF00" ratio="1"/>
        </ilog:gradientEntries>
      </ilog:CircularBarRenderer>
      <ilog:CircularBarRenderer
        radius="25%"
        value="95"
        placement="inside"
        editable="true">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0x0FF000" ratio="0"/>
          <mx:GradientEntry color="0xFF0000" ratio="1"/>
        </ilog:gradientEntries>
       </ilog:CircularBarRenderer>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

The CircularTrackRenderer class

A track renderer is a class that draws an annulus with arbitrary start and end angles, and arbitrary start and end thicknesses. It can be filled with a conical gradient or with a solid color. It is used on a scale to indicate that a value enters a given value range.

Customizing the geometry of a CircularTrackRenderer

The geometry of a `CircularTrackRenderer` is controlled by the following properties:

♦ `minimum` — Defines the value that corresponds to the start angle of the track.

♦ `maximum` — Defines the value that corresponds to the end angle of the track.

♦ `startThickness` — The thickness of the track at the start angle of the track.

♦ `endThicknessendThickness` — The thickness of the track at the end angle of the track.

♦ `placement` — The position of the annulus relative to the radius. Possible values are: `inside` (default), `outside`, and `cross`.

`startThickness` and `endThickness` can be expressed in a percentage of the radius or in pixels.

Customizing the colors of a CircularTrackRenderer

The `CircularTrackRenderer` supports two fill modes:

♦ Solid color — This is the default mode. The style properties `rendererColor` and `rendererAlpha` are used to fill the track.

♦ Color gradients — Fill the `gradientEntries` array with `GradientEntry` instances to obtain a conical gradient.

The following code example shows different scenarios of track renderers:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.graphics.*;
    ]]>
  </mx:Script>
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer/>
      <ilog:CircularTrackRenderer
        radius="25%"
        minimum="0"
        maximum="50"
        placement="outside"
        rendererColor="0x0000FF"
        stroke="{new Stroke(0xAAAAAA, 5)}" />
      <ilog:CircularTrackRenderer
        radius="25%"
        minimum="50"
        maximum="100"
        placement="inside"
        startThickness="0%">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0xFF0000" ratio="0"/>
          <mx:GradientEntry color="0x00FF00" ratio="1"/>
        </ilog:gradientEntries>
      </ilog:CircularTrackRenderer>
      <ilog:CircularTrackRenderer
        radius="15%"
        minimum="10"
        maximum="90"
        startThickness="40%" endThickness="40%"
        placement="inside">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0x0FF000" ratio="0"/>
          <mx:GradientEntry color="0xFF0000" ratio="1"/>
        </ilog:gradientEntries>
       </ilog:CircularTrackRenderer>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

The CircularLabelRenderer class

This class is used to render text in circular gauges. The position of the text is controlled by the following properties:

- `originX` and `originY` — The horizontal and vertical origins of the text.
- `horizontalAlign` — The horizontal placement of the text relative to `originX`. Possible values are: `left`, `center`, and `right`.
- `verticalAlign` — The vertical placement of the text relative to `originY`. Possible values are: `top`, `middle`, and `bottom`.

The text size is controlled by the property `labelFontSize`. It can be expressed in a percentage of the radius or in pixels.

The following code example shows a gauge with different label configurations:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircularScaleRenderer/>
      <ilog:CircleRenderer>
        <ilog:fill>
          <mx:SolidColor color="0xFFFF00"/>
        </ilog:fill>
        <ilog:stroke>
          <mx:Stroke color="0xDDDDDD"/>
        </ilog:stroke>
      </ilog:CircleRenderer>
      <ilog:CircularLabelRenderer text="Center" labelFontSize="20%"/>
      <ilog:CircularLabelRenderer text="Top Left"
        horizontalAlign="left" verticalAlign="top"
        labelFontSize="5%" originX="25%" originY="25%"/>
      <ilog:CircularLabelRenderer text="Bottom Left"
        horizontalAlign="left" verticalAlign="bottom"
        labelFontSize="5%" originX="25%" originY="75%"/>
      <ilog:CircularLabelRenderer text="Top Right"
        horizontalAlign="right" verticalAlign="top"
        labelFontSize="5%" originX="75%" originY="25%"/>
      <ilog:CircularLabelRenderer text="Bottom Right"
        horizontalAlign="right" verticalAlign="bottom"
        labelFontSize="5%" originX="75%" originY="75%"/>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

You can also define a custom renderer for rendering the label. The following example shows how to use an `AlphaNumericIndicator35` renderer to render the label.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircleRenderer>
        <ilog:fill>
          <mx:SolidColor color="0x000000"/>
        </ilog:fill>
      </ilog:CircleRenderer>
      <ilog:CircularScaleRenderer color="0xFFFFFF"/>
      <ilog:CircularLabelRenderer text="Center">
        <ilog:labelRenderer>
          <ilog:AlphaNumericIndicator35 numChars="0"/>
        </ilog:labelRenderer>
      </ilog:CircularLabelRenderer>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Note that if you redefine the renderer, the `labelFontSize` property is ignored.

The CircleRenderer class

The `CircleRenderer` class is a basic circle renderer with fill and stroke style properties. It is typically used for backgrounds or needle caps.

The following code example shows how to use Flex® filters, like `BevelFilter` and `DropShadowFilter`, to programmatically create the background of a gauge.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:CircleRenderer>
        <ilog:fill>
          <mx:SolidColor color="0x333333"/>
        </ilog:fill>
        <ilog:stroke>
          <mx:Stroke color="0xEEEEEE"/>
        </ilog:stroke>
        <ilog:filters>
          <mx:Array>
            <mx:BevelFilter quality="15" distance="2"/>
            <mx:DropShadowFilter quality="15" distance="10"
                                 color="0xAAAAAA" angle="90"/>
          </mx:Array>
        </ilog:filters>
      </ilog:CircleRenderer>
      <ilog:CircleRenderer radius="48%">
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xAAAAAA"/>
            <mx:GradientEntry color="0xEEEEEE"/>
          </mx:LinearGradient>
        </ilog:fill>
        <ilog:filters>
          <mx:Array>
            <mx:BevelFilter quality="15" distance="2"/>
          </mx:Array>
        </ilog:filters>
      </ilog:CircleRenderer>
      <ilog:CircularScaleRenderer radius="45%"/>
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

The CircularGaugeAsset class

The `CircularGaugeAsset` class allows you to add external assets to a circular gauge. It can be used to display static assets, like backgrounds, or assets that will be rotated, like needles.

The `mode` property defines whether the asset is a static decoration (`decoration`) that must not change with the value of the `CircularGaugeAsset` or whether the asset must rotate according to the value of the renderer (`rotation`).

The `CircularGaugeAsset` defines:

♦ The `assetOriginX` and `assetOriginY` properties that represent the pivot of the asset. These properties can be expressed in a percentage of the asset's original size or in pixels.

♦ The `offsetAngle` property that allows you to apply an additional offset to the asset rotation. Indeed, for the rotation of the asset to be correct with respect to the renderer value in `rotation` mode, the asset must be drawn horizontally from left to right.

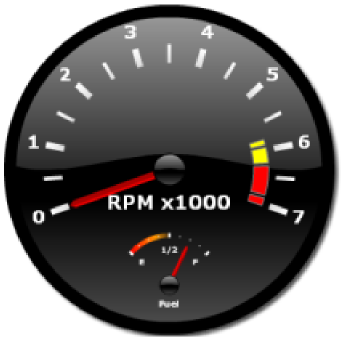
  For example: a designer has provided a needle asset drawn vertically from top to bottom. To use this asset as a needle which points to the renderer value in `rotation` mode, you need to adjust its angle by setting the `offsetAngle` to 90.

The following code example shows how to use an asset oriented from left to right as a needle.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:CircularGauge width="100%" height="100%">
    <ilog:scales>
      <ilog:CircularLinearScale/>
    </ilog:scales>
    <ilog:elements>
    <ilog:CircularScaleRenderer/>
      <ilog:CircularGaugeAsset
        assetOriginX="0%"
        mode="rotation"
        asset="@Embed(source='needle.swf')"
        radius="50%" radiusRelativeTo="width" value="20" />
    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Building a circular gauge step by step

This step-by-step tutorial shows how to build the following car-style circular gauge in MXML, with the IBM ILOG Elixir Gauge Framework.



This gauge is made up of the following elements:

- Three instances of `CircularGaugeAsset` with the same asset embedded: one for the background and two for needle caps
- Two instances of `CircularLinearScale`: one for RPM and one for Fuel
- Two instances of `CircularScaleRenderer` , one for each scale
- Two instances of `NeedleRenderer` , one for each scale
- Five instances of `CircularTrackRenderer`: one gradient track for the fuel scale and four for the RPM scale (two yellow ones, two red ones)
- Two instances of `CircularLabelRenderer`: one displays “RPM x 1000” and the other “Fuel”

You will configure each of these objects to design the gauge.

Configuring the logical scales

The logical scales are defined in the scales array. In this example, the two scales are circular linear scales.

To configure the logical scales:

- Write the following code:

```
    <ilog:scales>
      <ilog:CircularLinearScale minimum="0" maximum="7"
        startAngle="200" endAngle="340"
        majorTickInterval="1" minorTickInterval="0.5"
        snapInterval="0.5" id="rpm"/>
      <ilog:CircularLinearScale minimum="0" maximum="1"
        startAngle="140" endAngle="40"
        majorTickInterval="0.5" minorTickInterval="0.125"
        id="fuel" />
    </ilog:scales>
```

These scales are used by most visual elements, such as scale renderers, track renderers, and needle renderers.

Configuring the visual elements

The visual elements are defined in the elements array. They are drawn in the order they are in the array.

To configure the visual elements:

1. Define the black external assets.

   a. Define the background asset, which is the first element of the array.

   ```
       <ilog:elements>
         <!-- First, we add a vector graphic asset background with
              a drop shadow -->
         <ilog:CircularGaugeAsset
              asset="@Embed(source='black_circle.swf')">
           <ilog:filters>
             <mx:DropShadowFilter color="0x555555"/>
           </ilog:filters>
         </ilog:CircularGaugeAsset>
   ```

   Like any visual Flex® object, you can apply Flex filters to `CircularGaugeAsset`. In this example, a drop shadow is applied on the gauge background.

   b. Define the asset for the RPM needle cap, so that it is centered with a radius of 5%:

   ```
   <ilog:CircularGaugeAsset asset="@Embed(source='black_circle.swf')"
                            radius="5%"/>
   ```

   c. Define the asset for the Fuel needle cap with a radius of 3%, centered relatively to the width (the default value), and at 85% of the height of the gauge:

   ```
   <ilog:CircularGaugeAsset asset="@Embed(source='black_circle.swf')"
           originY="85%" radius="3%"/>
   ```

2. Define the scale renderers.

   There are two scale renderers, each referencing one of the logical scales.

   a. Define the RPM scale renderer like this:

   ```
       <ilog:CircularScaleRenderer scale="{rpm}" radius="38%"
             labelRadius="102%" labelPlacement="outside"
             majorTickLength="15%" minorTickLength="15%"
           labelFontSize="15%" fontWeight="bold">
           <ilog:majorTickRenderer>
             <mx:Component>
               <ilog:RectangleTickRenderer>
                 <ilog:fill>
                   <mx:LinearGradient angle="90">
   ```

```
              <mx:GradientEntry color="0xFFFFFF"/>
              <mx:GradientEntry color="0xDDDDDD"/>
            </mx:LinearGradient>
          </ilog:fill>
        </ilog:RectangleTickRenderer>
      </mx:Component>
    </ilog:majorTickRenderer>
    <ilog:minorTickRenderer>
      <mx:Component>
        <ilog:RectangleTickRenderer>
          <ilog:fill>
            <mx:LinearGradient angle="90">
              <mx:GradientEntry color="0xFFFFFF"/>
              <mx:GradientEntry color="0xDDDDDD"/>
            </mx:LinearGradient>
          </ilog:fill>
        </ilog:RectangleTickRenderer>
      </mx:Component>
    </ilog:minorTickRenderer>
  </ilog:CircularScaleRenderer>
```

b. Define the Fuel scale renderer like this:

```
  <ilog:CircularScaleRenderer scale="{fuel}" originY="85%"
    tickPlacement="cross" fontWeight="bold" radius="15%"
    labelFunction="fuelLabel" labelFontSize="15%">
    <ilog:majorTickRenderer>
      <mx:Component>
        <ilog:RectangleTickRenderer>
          <ilog:fill>
            <mx:SolidColor color="white"/>
          </ilog:fill>
        </ilog:RectangleTickRenderer>
      </mx:Component>
    </ilog:majorTickRenderer>
    <ilog:minorTickRenderer>
      <mx:Component>
        <ilog:CircleTickRenderer>
          <ilog:fill>
            <mx:SolidColor color="white"/>
          </ilog:fill>
        </ilog:CircleTickRenderer>
      </mx:Component>
    </ilog:minorTickRenderer>
  </ilog:CircularScaleRenderer>
```

3. Define the needles.

In this gauge, there are two needles. If the `scale` property is not set, the first scale is used. Use the properties `editable`, `editMode`, `mouseMode`, and `animationDuration` to configure the interaction with the needle.

```
  <!-- The RPM needle -->
  <ilog:NeedleRenderer thickness="7%" radius="32%" editable="true"
    editMode="discrete" mouseMode="area" animationDuration="200">
```

```
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xEE0000" ratio="0.3"/>
            <mx:GradientEntry color="0xAA0000" ratio="0.5"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:NeedleRenderer>
      <!-- The fuel needle -->
      <ilog:NeedleRenderer thickness="8%" scale="{fuel}"
        originY="85%" radius="13%" value="0.75"
        editable="true" mouseMode="area">
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xEE0000" ratio="0.3"/>
            <mx:GradientEntry color="0xAA0000" ratio="0.5"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:NeedleRenderer>
```

4. Define the tracks.

They are used to render a customizable annulus between two scale values.

a. The RPM gauge has four tracks with solid colors defined as follows:

```
      <!-- Four track renderers of the RPM Gauge -->
      <ilog:CircularTrackRenderer minimum="5.8" maximum="5.9"
        rendererColor="yellow" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="5.95" maximum="6.25"
        rendererColor="yellow" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="6.3" maximum="6.85"
        rendererColor="red" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="6.9" maximum="7.05"
        rendererColor="red"
        radius="30%" startThickness="15%" endThickness="15%"/>
```

b. The Fuel gauge has one gradient track defined as follows:

```
      <!-- The Fuel gradient track -->
      <ilog:CircularTrackRenderer minimum="0" maximum="0.5"
       scale="{fuel}" originY="85%" radius="15%" startThickness="10%"

        endThickness="10%" placement="cross">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0xFF0000" ratio="0"/>
          <mx:GradientEntry color="0xFF9900" ratio="0.7" alpha="1"/>
          <mx:GradientEntry color="0xFF9900" ratio="0.8"
              alpha="0.7"/>
          <mx:GradientEntry color="0xFF9900" ratio="0.9"
              alpha="0.5"/>
          <mx:GradientEntry color="0xFF9900" ratio="1" alpha="0.2"/>
        </ilog:gradientEntries>
      </ilog:CircularTrackRenderer>
```

Following is the complete code of this tutorial:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex" backgroundColor="0xFFFFFF">

  <mx:Script>
    <![CDATA[
      import ilog.gauges.TickItem;

      // This sample shows how to create a car-style gauge with RPM
      // and Fuel indicators.

      // A Label function to customize labels of the fuel gauge.
      private function fuelLabel(t:TickItem):String
      {
        switch(t.value)
        {
          case 0: return "E";     // Empty tank
          case 0.5: return "1/2";
          case 1: return "F";     // Full tank
          default: return "";
        }
      }
    ]]>
  </mx:Script>

  <!-- We set labels color to white -->
  <ilog:CircularGauge width="80%" height="80%" color="0xFFFFFF"
     backgroundColor="0xFFFFFF">

    <!-- Logical scales definition, one for the RPM gauge the other
         for the fuel gauge -->
    <ilog:scales>
      <ilog:CircularLinearScale minimum="0" maximum="7"
        startAngle="200" endAngle="340"
        majorTickInterval="1" minorTickInterval="0.5"
        snapInterval="0.5" id="rpm"/>
      <ilog:CircularLinearScale minimum="0" maximum="1"
        startAngle="140" endAngle="40"
        majorTickInterval="0.5" minorTickInterval="0.125" id="fuel" />
    </ilog:scales>

    <!-- The array of visual elements -->
    <ilog:elements>
      <!-- First, we add a vector graphic asset background with a
           drop shadow -->
      <ilog:CircularGaugeAsset
          asset="@Embed(source='black_circle.swf')">
        <ilog:filters>
          <mx:DropShadowFilter color="0x555555"/>
        </ilog:filters>
      </ilog:CircularGaugeAsset>

      <!-- The RPM scale renderer -->
      <ilog:CircularScaleRenderer scale="{rpm}" radius="38%"
        labelRadius="102%" labelPlacement="outside"
        majorTickLength="15%" minorTickLength="15%"
        labelFontSize="15%" fontWeight="bold">
        <ilog:majorTickRenderer>
          <mx:Component>
            <ilog:RectangleTickRenderer>
              <ilog:fill>
                <mx:LinearGradient angle="90">
                  <mx:GradientEntry color="0xFFFFFF"/>
                  <mx:GradientEntry color="0xDDDDDD"/>
                </mx:LinearGradient>
              </ilog:fill>
            </ilog:RectangleTickRenderer>
          </mx:Component>
        </ilog:majorTickRenderer>
        <ilog:minorTickRenderer>
          <mx:Component>
            <ilog:RectangleTickRenderer>
              <ilog:fill>
                <mx:LinearGradient angle="90">
                  <mx:GradientEntry color="0xFFFFFF"/>
                  <mx:GradientEntry color="0xDDDDDD"/>
                </mx:LinearGradient>
              </ilog:fill>
            </ilog:RectangleTickRenderer>
          </mx:Component>
        </ilog:minorTickRenderer>
      </ilog:CircularScaleRenderer>

      <!-- Four track renderers of the RPM Gauge -->
      <ilog:CircularTrackRenderer minimum="5.8" maximum="5.9"
        rendererColor="yellow" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="5.95" maximum="6.25"
        rendererColor="yellow" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="6.3" maximum="6.85"
        rendererColor="red" radius="30%" startThickness="15%"
        endThickness="15%"/>
      <ilog:CircularTrackRenderer minimum="6.9" maximum="7.05"
        rendererColor="red" radius="30%" startThickness="15%"
        endThickness="15%"/>

      <!-- The "RPM x1000" label -->
      <ilog:CircularLabelRenderer text="RPM x1000" originY="60%"
        labelFontSize="12%" fontWeight="bold"/>

      <!-- The RPM needle -->
      <ilog:NeedleRenderer thickness="7%" radius="32%" editable="true"
```

```
    editMode="discrete" mouseMode="area" animationDuration="200">
    <ilog:fill>
      <mx:LinearGradient angle="90">
        <mx:GradientEntry color="0xEE0000" ratio="0.3"/>
        <mx:GradientEntry color="0xAA0000" ratio="0.5"/>
      </mx:LinearGradient>
    </ilog:fill>
  </ilog:NeedleRenderer>

  <!-- The RPM needle cap -->
  <ilog:CircularGaugeAsset
      asset="@Embed(source='black_circle.swf')" radius="5%"/>

  <!-- Fuel sub-gauge part -->

  <!-- The Fuel gradient track -->
  <ilog:CircularTrackRenderer minimum="0" maximum="0.5"
    scale="{fuel}" originY="85%" radius="15%"
    startThickness="10%" endThickness="10%" placement="cross">
    <ilog:gradientEntries>
      <mx:GradientEntry color="0xFF0000" ratio="0"/>
      <mx:GradientEntry color="0xFF9900" ratio="0.7" alpha="1"/>
      <mx:GradientEntry color="0xFF9900" ratio="0.8" alpha="0.7"/>
      <mx:GradientEntry color="0xFF9900" ratio="0.9" alpha="0.5"/>
      <mx:GradientEntry color="0xFF9900" ratio="1" alpha="0.2"/>
    </ilog:gradientEntries>
  </ilog:CircularTrackRenderer>

  <!-- The fuel scale renderer -->
  <ilog:CircularScaleRenderer scale="{fuel}" originY="85%"
    tickPlacement="cross" fontWeight="bold" radius="15%"
    labelFunction="fuelLabel" labelFontSize="15%">
    <ilog:majorTickRenderer>
      <mx:Component>
        <ilog:RectangleTickRenderer>
          <ilog:fill>
            <mx:SolidColor color="white"/>
          </ilog:fill>
        </ilog:RectangleTickRenderer>
      </mx:Component>
    </ilog:majorTickRenderer>
    <ilog:minorTickRenderer>
      <mx:Component>
        <ilog:CircleTickRenderer>
          <ilog:fill>
            <mx:SolidColor color="white"/>
          </ilog:fill>
        </ilog:CircleTickRenderer>
      </mx:Component>
    </ilog:minorTickRenderer>
  </ilog:CircularScaleRenderer>

  <!-- The fuel needle -->
  <ilog:NeedleRenderer thickness="8%" scale="{fuel}"
```

```
        originY="85%" radius="13%" value="0.75"
        editable="true" mouseMode="area">
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xEE0000" ratio="0.3"/>
            <mx:GradientEntry color="0xAA0000" ratio="0.5"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:NeedleRenderer>

      <!-- The fuel needle cap -->
      <ilog:CircularGaugeAsset
        asset="@Embed(source='black_circle.swf')"
        originY="85%" radius="3%"/>

      <!-- "Fuel" label -->
      <ilog:CircularLabelRenderer text="Fuel" originY="91%"
        labelFontSize="5%" fontWeight="bold"/>

    </ilog:elements>
  </ilog:CircularGauge>
</mx:Application>
```

Custom rectangular gauges

Describes the elements that make up a rectangular gauge and how to configure them to create a rectangular gauge from scratch.

In this section

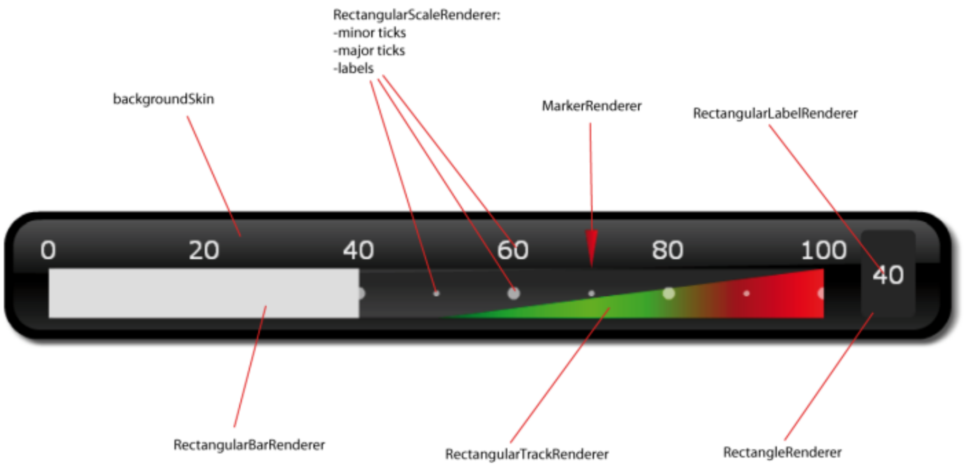
Overview of a rectangular gauge
Illustrates the elements that compose a rectangular gauge.

The logical scale(s) of rectangular gauges
The first thing to do when creating a custom gauge is to define the scale(s) it will be using.

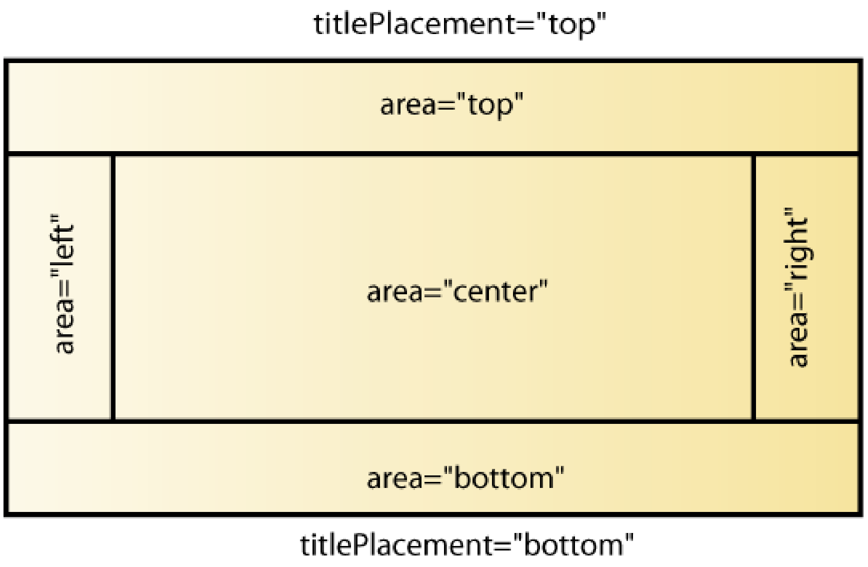
The visual elements of rectangular gauges
Describes the main visual elements of rectangular gauges and how to customize them. Information on `RectangularGaugeWrapper` is in the reference manual only.

Building a rectangular gauge step by step
Explains how to build a rectangular gauge from scratch.

Overview of a rectangular gauge



A rectangular gauge can be horizontal or vertical depending on the value of its `direction` property It can contain several visual elements, as shown in the illustration above. These elements are laid out according to the value of their `area` property as shown in the following illustration:



The side areas (top, bottom, left, and right) are used to place fixed size indicators, such as labels or various indicators about the gauge state. The center of the gauge is used for the core elements, such as the scale or the value renderers.

The visual elements of a rectangular gauge extend the `RectangularGaugeElement` class or one of its subclasses. They are characterized by:

- An area of reference that can either be `top`, `bottom`, `left`, `right`, or `center`
- A position and a size specified using `x`, `y`, `width`, and `height` properties, optionally expressed in a percentage of their corresponding area

The layout of the visual elements of a rectangular gauge is processed in the following order:

1. If the `showTitle` style property is `true`, the title is placed at a reserved area at the top or bottom depending on the value of the `titlePlacement` property.
2. The side areas are sized so that the elements in the left and right, on one hand, and top and bottom, on the other hand, can fit respectively their measured width and height inside these areas.
3. The elements in the side areas are positioned and sized according to the value of their `x`, `y`, `width`, and `height` properties. Percentage values are relative to the bounds of the corresponding areas computed in step 2.
4. The remaining area in the center of the gauge is used to lay out the elements in this area. If the gauge has no explicit width, either in pixels or in percentage, the width of the center area will be computed from the measured width of the elements in this area. Similarly, if the gauge has no explicit height, either in pixels or in percentage, the height of the center area will be computed from the measured height of the elements in this area. If the gauge has an explicit width (or height), the elements in the center take as much width (or height) as is available
5. The background and foreground skins, defined using the `backgroundSkin` and `foregroundSkin` properties, if any, are sized so that they cover the whole gauge except the title area.

The logical scale(s) of rectangular gauges

A scale is a logical element that handles the main characteristics of a gauge: minimum and maximum values (numeric scales), categories (category scales), length and offset of reference. A scale can be numeric or category-based. Scales are used by many visual elements to render themselves. For example, the `RectangularScaleRenderer` or the `MarkerRenderer` use their associated scale to render themselves.

Available rectangular scales are:

- `RectangularLinearScale`
- `RectangularLogScale`
- `RectangularCategoryScale`

The following example defines a logical scale with its associated scale renderer and a marker pointing a specific value.

```
<ilog:RectangularGauge width="100%">
  <ilog:scales>
    <ilog:RectangularLinearScale
      minimum="10" maximum="100"
      majorTickInterval="10" minorTickInterval="5"/>
  </ilog:scales>
  <ilog:elements>
    <ilog:RectangularScaleRenderer id="sr"/>
    <ilog:RectangularBarRenderer value="30"
     height="{sr.tickAreaBounds.height}"/>
  </ilog:elements>
</ilog:RectangularGauge>
```

If the scale attribute is not specified on a visual element, the visual element is automatically associated with the first logical scale in the scales array. If you need to reference a different scale, use the `scale` property.

A scale provides two arrays of `TickItem`, one for major ticks and one for minor ticks, that are used by the `RectangularScaleRenderer` to position the ticks. By default, the arrays are automatically computed, but you can override the default behavior by setting the scale properties, for example, the desired interval between two ticks.

Note: RectangularCategoryScale and RectangularLogScale do not generate minor tick items.

The length and offset of reference are usually computed by the scale master so that you do not have to set them. The scale master is the visual element that is used as a reference by the scale to do its `positionForValue` and `valueForPosition` computations. Usually, and by default, this is the unique or last `RectangularScaleRenderer` of the gauge. A visual element inheriting from `RectangularScaleRelatedRenderer` can ask to be the master of its scale by setting the `master` property to `true`. However, you can only have a single master for a given scale. Once a master has set the length and offset of a logical scale, all subsequent calls to the `positionForValue` method of the scale will take that length and offset into account. For example, if `positionForValue` is called with the minimum value of the scale, the offset position will be returned. If the `positionForValue` method is called with the maximum value of the scale, the offset+length position will be returned. This allows all visual elements to share the same transformation from the data space of the scale to the pixel space.

The visual elements of rectangular gauges

Describes the main visual elements of rectangular gauges and how to customize them. Information on `RectangularGaugeWrapper` is in the reference manual only.

In this section

The RectangularScaleRenderer class
Describes the `RectangularScaleRenderer` class and gives code examples.

About value renderer classes
Introduces the value renderer classes: `NeedleRenderer`, `CircularBarRenderer`, and `CircularGaugeAsset`.

The MarkerRenderer class
Describes the `MarkerRenderer` class and gives code examples.

The RectangularBarRenderer class
Describes the `RectangularBarRenderer` class and gives code examples.

The RectangularTrackRenderer class
Describes the `RectangularTrackRenderer` class and gives code examples.

The RectangularLabelRenderer class
Describes the `RectangularLabelRenderer` class and gives code examples.

The RectangleRenderer class
Describes the `RectangleRenderer` class and gives code examples.

The RectangularGaugeAsset class

Describes the `RectangularGaugeAsset` class and gives code examples.

The RectangularScaleRenderer class

The scale renderer is the visual element responsible for rendering the ticks and labels of a scale. The `RectangularScaleRenderer` class uses data provided by the logical scale, like `majorTicks` and `minorTicks` arrays.

The scale render is usually considered as the logical scale master. In other words, all the visual elements that use the scale to render themselves will transform the data to pixel coordinates in the same way as the scale renderer.

The scale renderer is made of two areas:

- ♦ The label area. This area is sized so that the labels can fit in it. It can be on the top or bottom side of the scale renderer for horizontal gauges, or on the left or right side of the scale renderer for vertical gauges. This choice is defined through the `labelPlacement` property of the scale renderer.

- ♦ The tick area. This area takes up the remaining area of the scale renderer. Its bounds are defined in the `tickAreaBounds` property of the scale renderer. Usually, other visual elements, such as `MarkerRenderer` or `RectangularBarRenderer` use these bounds to lay out themselves so that they are drawn on the tick area and not on the whole scale renderer area. See *Building a rectangular gauge step by step*.

Formatting tick marks

You can customize the rendering of the `RectangularScaleRenderer` ticks by setting the `majorTickRenderer` property for major ticks and the `minorTickRenderer` property for minor ticks. If the specified renderer implements `IDataRenderer`, the associated `TickItem` is set to the data property of the renderer.

The gauges framework includes two default tick item renderers: `RectangleTickRenderer` and `CircleTickRenderer`. The size of the tick renderer instances is controlled by the properties `majorTickWidth`, `majorTickLength`, `minorTickWidth`, and `minorTickLength`. These properties can all be expressed in pixels or in a percentage of the tick area of the scale renderer.

The placement of the tick renderer instances is based on the value of the `tickPlacement` property. Possible values are: `leading`, `trailing`, or `middle`.

The following example shows how to format the tick marks of a `RectangularScaleRenderer` and change the renderer for major ticks:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:RectangularGauge>
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer tickPlacement="trailing"
        majorTickWidth="2%" majorTickLength="50%"
        minorTickWidth="1%" minorTickLength="20%">
        <ilog:majorTickRenderer>
          <mx:Component>
            <ilog:RectangleTickRenderer>
```

```
              <ilog:fill>
                <mx:SolidColor color="0xFF0000"/>
              </ilog:fill>
            </ilog:RectangleTickRenderer>
          </mx:Component>
        </ilog:majorTickRenderer>
      </ilog:RectangularScaleRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

Formatting labels

You can size and format the scale labels using the properties `labelPlacement` and `labelFunction`, as well as the various style properties related to fonts.

The following example shows how to place the labels at the leading edge of the scale (on the left side since the gauge is vertical) with a customized text, a specific font size, family, and color:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.gauges.TickItem;
      private function withDegrees(t:TickItem):String {
        return t.value + "°";
      }
    ]]>
  </mx:Script>
  <ilog:RectangularGauge height="100%" direction="vertical">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer fontSize="24" color="red"
        fontFamily="Arial" labelPlacement="leading"
        labelFunction="{withDegrees}" />
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

Defining a custom renderer

You can also define a custom renderer for rendering labels of the scale. The following example shows how to use a `NumericIndicator` renderer to render the labels.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0x000000">
  <ilog:RectangularGauge>
    <ilog:scales>
      <ilog:RectangularLinearScale minimum="100" maximum="150"
majorTickInterval="10" minorTickInterval="5"/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer>
        <ilog:labelRenderer>
          <mx:Component>
            <ilog:NumericIndicator numChars="3" textHorizontalAlign="right"/>

          </mx:Component>
        </ilog:labelRenderer>
      </ilog:RectangularScaleRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

About value renderer classes

A value renderer is a renderer class that represents a data value. The rectangular gauges framework provides three predefined value renderers:

- `MarkerRenderer` — A customizable shape that points the value of the renderer. See *The MarkerRenderer class* for details.
- `RectangularBarRenderer` — A customizable bar that starts from the start position of the scale and extends to the position corresponding to the value of the renderer. See *The RectangularBarRenderer class* for details.
- `RectangularGaugeAsset` — A value renderer that delegates the rendering to an `IFlexDisplayObject`, which usually represents an external asset. See *The RectangularGaugeAsset class* for details.

These renderers extend the `RectangularValueRenderer` class, which can be customized using the following properties:

- `value` — The value the renderer represents.
- `editable` — This property indicates whether the value can be edited using mouse gestures.
- `liveDragging` — If this property is set to true, the value of the renderer changes during user interaction and not only after the interaction (which is the default behavior).
- `editMode` — The possible values of this property are: `discrete` and `continuous`. If the property is set to `discrete`, the value renderer snaps to the scale accepted values (categories for `RectangularCategoryScale`, or value on the snapping interval for `RectangularLinearScale`) when the user moves the mouse. Otherwise, the rendered value moves continuously.
- `mouseMode` — This property defines the reactive area of the renderer during an interaction. If it is set to `shape`, only the drawn area of the renderer reacts to the mouse. Otherwise, the reactive area corresponds to the bounds of the renderer.

The MarkerRenderer class

The marker renderer is a visual element responsible for rendering the marker of the gauge. It references a logical scale to point to the position defined by its `value` property using a specific shape. A marker can also be represented by an external asset, see *The RectangularGaugeAsset class*.

Customizing the geometry of a MarkerRenderer

The thickness of the marker shape is determined by the `thickness` property, expressed in pixels or in a percentage of the renderer width, if the gauge is horizontal, or height, if the gauge is vertical.

The length of the marker shape is determined by the `length` property, expressed in pixels or in a percentage of the renderer height, if the gauge is horizontal, or width, if the gauge is vertical.

The placement of the marker shape inside the marker bounds is determined by the `placement` property which can be set to `leading`, `trailing`, or `middle`.

A marker renderer can have a rectangular shape or an arrow shape. This is defined using the `form` style property of the renderer.

When the marker renderer has an arrow shape, it is drawn as an arrow pointing to the direction which corresponds to its placement. If the placement is `middle` then a dual arrow pointing in both directions is drawn. For example, if the gauge is horizontal and the `MarkerRenderer` placement is `trailing`, then the arrow is drawn at the bottom, pointing to the bottom direction. If the gauge is horizontal and the `MarkerRenderer` placement is `leading`, then the arrow is drawn at the left side, pointing to the left direction.

You can subclass the `MarkerRenderer` class to redefine its `drawMarker` method and draw another kind of marker.

The following example shows two options to customize the marker geometry.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.graphics.SolidColor;
    ]]>
  </mx:Script>
  <ilog:RectangularGauge width="100%" height="100">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer showLabels="false" height="50%"/>
      <ilog:MarkerRenderer y="50%" height="50%" value="0"
        thickness="2%"
        placement="leading" form="arrow"
        editable="true" mouseMode="shape"
        fill="{new SolidColor(0xAA0000)}" />
      <ilog:MarkerRenderer value="20" height="50%"
```

```
        length="80%" thickness="10"
        editable="true" mouseMode="shape"
        fill="{new SolidColor(0xAAAA00)}" />
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

The fill and stroke of a marker can also be changed with the `fill` and `stroke` style properties.

The RectangularBarRenderer class

This class represents a bar whose position extends from the minimum value of the associated scale to the value of the renderer.

The geometry of a `RectangularBarRenderer` is controlled by the following properties:

- ♦ `rounded` — Indicates whether the corners of the bar are rounded or not.
- ♦ `radius` — Defines the radius of the rounded corners, if the corners are rounded.

The fill and stroke of the bar can be changed with the `fill` and `stroke` style properties. The fill of the bar renderer is drawn so that its definition rectangle extends from the minimum value of the associated scale to the maximum value. For example, if the fill is a gradient from green to red, the red color is visible only when the maximum value of the scale is reached. For a lower value, the gradient will stop at the corresponding color.

The following code example shows different scenarios of bar renderers

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.graphics.*;
    ]]>
  </mx:Script>
  <ilog:RectangularGauge height="100%" direction="vertical">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer x="75%" width="25%"/>
      <ilog:RectangularBarRenderer x="2%" width="21%" value="100"
        editable="true"
        fill="{new SolidColor(0x0000FF)}"
        stroke="{new Stroke(0xAAAAAA, 5)}"/>
      <ilog:RectangularBarRenderer x="27%" width="21%" value="80"
        editable="true" rounded="true" radius="10">
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xFF0000" ratio="0"/>
            <mx:GradientEntry color="0x00FF00" ratio="1"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:RectangularBarRenderer>
      <ilog:RectangularBarRenderer x="52%" width="21%" value="95"
        editable="true">
        <ilog:fill>
          <mx:LinearGradient angle="90">
             <mx:GradientEntry color="0x0FF000" ratio="0"/>
             <mx:GradientEntry color="0xFF0000" ratio="1"/>
          </mx:LinearGradient>
        </ilog:fill>
       </ilog:RectangularBarRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

The RectangularTrackRenderer class

A track renderer is a class that draws a range of arbitrary values with arbitrary start and end thicknesses. It can be filled with a gradient or with a solid color. It is used on a scale to indicate that a value enters a given value range.

Customizing the geometry of a RectangularTrackRenderer

The geometry of a `RectangularTrackRenderer` is controlled by the following properties:

- `minimum` — Defines the value that corresponds to the start position of the track.
- `maximum` — Defines the value that corresponds to the end position of the track.
- `startThickness` — The thickness of the track at the start position.
- `endThickness` — The thickness of the track at the end position.
- `placement` — The position of the track relative to its bounds.

`startThickness` and `endThickness` can be expressed in pixels or as a percentage of the renderer's height (horizontal gauges) or renderer's width (vertical gauges).

Customizing the colors of a RectangularTrackRenderer

The `RectangularTrackRenderer` supports two fill modes:

- Solid color — This is the default mode. The style properties `rendererColor` and `rendererAlpha` are used to fill the track.
- Color gradients — Fill the `gradientEntries` array with `GradientEntry` instances to obtain a linear gradient extending from the start position to the end position.

The following code example shows different scenarios of track renderers:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <mx:Script>
    <![CDATA[
      import mx.graphics.*;
    ]]>
  </mx:Script>
  <ilog:RectangularGauge width="100%" height="150">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer y="33%" height="33%"
         showLabels="false"/>
      <ilog:RectangularTrackRenderer y="0%" height="30%"
        minimum="0" maximum="50"
        placement="trailing"
        rendererColor="0x0000FF" stroke="{new Stroke(0xAAAAAA, 2)}"/>
      <ilog:RectangularTrackRenderer y="69%" height="30%"
        minimum="50" maximum="100"
        placement="leading" startThickness="33%">
        <ilog:gradientEntries>
          <mx:GradientEntry color="0xFF0000" ratio="0"/>
          <mx:GradientEntry color="0x00FF00" ratio="1"/>
        </ilog:gradientEntries>
      </ilog:RectangularTrackRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

The RectangularLabelRenderer class

This class is used to render text in rectangular gauges. The text is usually placed on a side area of the gauge (see *Overview of a rectangular gauge*). The position of the text is controlled by:

- ♦ The bounds of the element.
- ♦ `horizontalAlign` — The horizontal placement of the text relative to the bounds width. Possible values are: `left`, `center`, and `right`.
- ♦ `verticalAlign` — The vertical placement of the text relative to the bounds height. Possible values are: `top`, `middle`, and `bottom`.

The styling of the text is controlled by the font style properties like `fontSize` and `fontFamily`.

The following code example shows a gauge with different label configurations:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge width="100%">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularLabelRenderer text="Center" fontSize="20"
        area="center"/>
      <ilog:RectangularLabelRenderer text="Top"
        verticalAlign="top"
        fontSize="15" area="top"/>
      <ilog:RectangularLabelRenderer text="Left"
        horizontalAlign="left"
        fontSize="15" area="left"/>
      <ilog:RectangularLabelRenderer text="Right"
        horizontalAlign="right"
        fontSize="15" area="right"/>
      <ilog:RectangularLabelRenderer text="Bottom"
        verticalAlign="bottom"
        fontSize="15" area="bottom"/>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

You can also define a custom renderer for rendering the label. The following example shows how to use an `AlphaNumericIndicator35` renderer to render the label.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge width="100%">
    <ilog:scales>
```

```
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer/>
      <ilog:RectangularLabelRenderer text="Top"
        verticalAlign="top" area="top">
        <ilog:labelRenderer>
         <ilog:AlphaNumericIndicator35 numChars="0" onSegmentColor="0x550000"
 offSegmentColor="0xCCCCCC"/>
        </ilog:labelRenderer>
      </ilog:RectangularLabelRenderer>
      <ilog:RectangularLabelRenderer text="Bottom"
        verticalAlign="bottom" area="bottom">
        <ilog:labelRenderer>
         <ilog:AlphaNumericIndicator35 numChars="0" onSegmentColor="0x550000"
 offSegmentColor="0xCCCCCC"/>
        </ilog:labelRenderer>
      </ilog:RectangularLabelRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

The RectangleRenderer class

This class is a basic rectangle renderer with fill and stroke style properties. It is typically used for backgrounds.

The following code example shows how to use Flex® filters, like `BevelFilter` and `DropShadowFilter`, to programmatically create the background of a gauge.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge width="100%">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangleRenderer>
        <ilog:fill>
          <mx:SolidColor color="0x333333"/>
        </ilog:fill>
        <ilog:stroke>
          <mx:Stroke color="0xEEEEEE"/>
        </ilog:stroke>
        <ilog:filters>
          <mx:Array>
            <mx:BevelFilter quality="15" distance="2"/>
            <mx:DropShadowFilter quality="15" distance="10"
                                 color="0xAAAAAA" angle="90"/>
          </mx:Array>
        </ilog:filters>
      </ilog:RectangleRenderer>
      <ilog:RectangleRenderer>
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry color="0xAAAAAA"/>
            <mx:GradientEntry color="0xEEEEEE"/>
          </mx:LinearGradient>
        </ilog:fill>
        <ilog:filters>
          <mx:Array>
            <mx:BevelFilter quality="15" distance="2"/>
          </mx:Array>
        </ilog:filters>
      </ilog:RectangleRenderer>
      <ilog:RectangularScaleRenderer/>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

The RectangularGaugeAsset class

This class allows you to add external assets to a rectangular gauge. It can be used to display static assets, like backgrounds, or assets that move, like markers, or that can be resized, like bars. The `mode` property defines whether the asset is a static decoration (`decoration`) that must not change with the value of the `RectangularGaugeAsset RectangularGaugeAsset`, or whether it must move as a marker (`marker`) or be reshaped as a bar (`bar`) according to the value of the renderer.

The following code example shows how to use an asset oriented from left to right as a needle.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:RectangularGauge width="100%">
    <ilog:scales>
      <ilog:RectangularLinearScale/>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer id="sr"/>
      <ilog:RectangularGaugeAsset asset="@Embed(source='marker.swf')"
                                  value="20"
                                  x="{sr.tickAreaBounds.x}"
                                  y="{sr.tickAreaBounds.y}"
                                  width="{sr.tickAreaBounds.width}"
                                  height="{sr.tickAreaBounds.height}"/>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

Building a rectangular gauge step by step

This section shows how to build a new rectangular gauge step by step.

To build a rectangular gauge step by step:

1. Create the gauge and size it.

   Usually you want horizontal gauges to have a fixed height and a proportional width and vertical gauges to have a fixed width and a proportional height. If width or height are not set, the gauge is displayed with its measured width or height.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge height="100%" direction="vertical">
  </ilog:RectangularGauge>
</mx:Application>
```

2. Decide on the type of scale(s) and add it (them) to the gauge.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge height="100%" direction="vertical">
    <ilog:scales>
      <ilog:RectangularCategoryScale>
        <ilog:categories>
          <mx:String>Low</mx:String>
          <mx:String>Medium</mx:String>
          <mx:String>High</mx:String>
        </ilog:categories>
      </ilog:RectangularCategoryScale>
    </ilog:scales>
  </ilog:RectangularGauge>
</mx:Application>
```

3. Create a renderer for the scale and customize it.

   In this particular case, each tick is drawn with a color according to the corresponding value.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge height="100%" direction="vertical">
    <ilog:scales>
      <ilog:RectangularCategoryScale>
        <ilog:categories>
          <mx:String>Low</mx:String>
          <mx:String>Medium</mx:String>
          <mx:String>High</mx:String>
        </ilog:categories>
      </ilog:RectangularCategoryScale>
    </ilog:scales>
    <ilog:elements>
      <ilog:RectangularScaleRenderer id="sr" majorTickWidth="2">
        <ilog:majorTickRenderer>
          <mx:Component>
            <ilog:RectangleTickRenderer>
              <ilog:fill>
                <mx:SolidColor color="{colorFromData(data)}"/>
              </ilog:fill>
              <mx:Script>
                <![CDATA[
                  private function colorFromData(data:Object):uint {
                    switch (data.value) {
                      case "Low":
                        return 0x00ff00;
                      case "Medium":
                        return 0x0000ff;
                      case "High":
                        return 0xff0000;
                    }
                    return 0x000000;
                  }
                ]]>
              </mx:Script>
            </ilog:RectangleTickRenderer>
          </mx:Component>
        </ilog:majorTickRenderer>
      </ilog:RectangularScaleRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
</mx:Application>
```

If no scale renderer is used, you must set another visual element as the logical scale master.

4. Create the renderer in charge of representing the gauge value and position it

   a. Make sure that the bar renderer covers only the tick area of the scale, not the whole scale area, by binding its coordinates to the coordinates of the tick area.

   b. Bind the value of the renderer to an external data source, a combo box in this case.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
  <ilog:RectangularGauge height="100%" direction="vertical">
    <!-- ... -->
    <ilog:elements>
      <!-- ... -->
      <ilog:RectangularBarRenderer id="vr" value="{combo.selectedItem}"
        x="{sr.tickAreaBounds.x}" y="{sr.tickAreaBounds.y}"
        width="{sr.tickAreaBounds.width}"
        height="{sr.tickAreaBounds.height}">
        <ilog:fill>
          <mx:LinearGradient angle="90">
            <mx:GradientEntry ratio="0" color="red"/>
            <mx:GradientEntry ratio="0.5" color="blue"/>
            <mx:GradientEntry ratio="1" color="green"/>
          </mx:LinearGradient>
        </ilog:fill>
      </ilog:RectangularBarRenderer>
    </ilog:elements>
  </ilog:RectangularGauge>
  <mx:ComboBox id="combo" selectedIndex="1">
    <mx:String>Low</mx:String>
    <mx:String>Medium</mx:String>
    <mx:String>High</mx:String>
  </mx:ComboBox>
</mx:Application>
```

5. Add elements that will not be in the center area, for example a label indicating the current value of the gauge.

```
   <ilog:elements>
      <ilog:RectangularLabelRenderer text="{vr.value}"
          area="top" fontSize="36" width="150" />
    </ilog:elements>
```

6. Add decoration elements, a background element under the label in this case.

```
   <ilog:elements>
      <ilog:RectangleRenderer area="top">
        <ilog:fill>
          <mx:RadialGradient>
            <mx:GradientEntry ratio="0" color="white" alpha="0.5"/>
            <mx:GradientEntry ratio="1" color="black" alpha="0.2"/>
          </mx:RadialGradient>
        </ilog:fill>
      </ilog:RectangleRenderer>
      <ilog:RectangularLabelRenderer .../>
    </ilog:elements>
```

7. Add other properties from the `RectangularGaugeAsset` `RectangularGauge` class, for example a title or background and foreground skins.

Indicators

Describes the use of IBM ILOG Elixir indicators with Adobe® Flex® 3.

In this section

Introduction to indicators
Describes the three kinds of indicators: 7–segment, 14–segment, and 35–dot, and their uses.

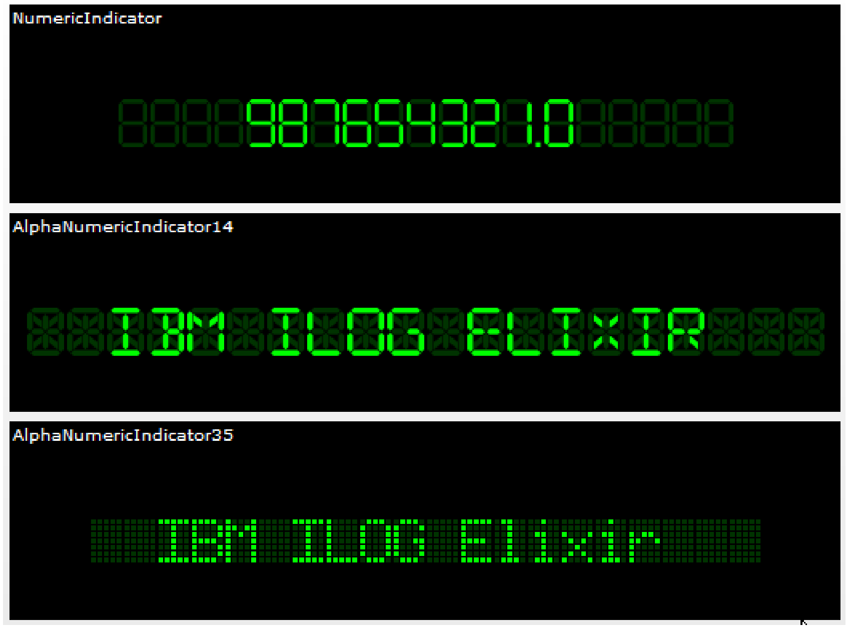
Indicators architecture
Describes the architecture of the indicators classes.

Using indicators
Describes how to use the predefined indicators.

Styling your indicators
Describes how to style the predefined indicators by changing style properties.

Introduction to indicators

IBM ILOG Elixir indicators display a single line of text with a choice of 7-segment, 14-segment and 35-dot character rendering. The following figure shows the three kinds of indicator display provided by IBM ILOG Elixir.

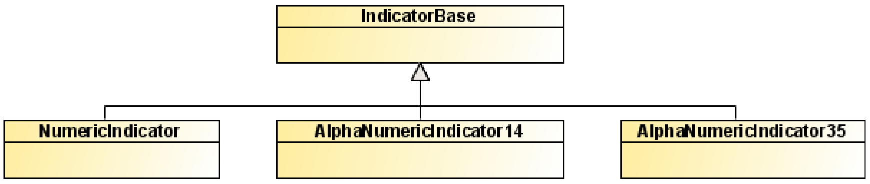


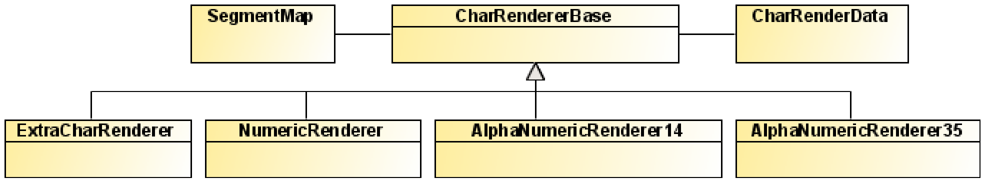
The styling of indicators is customizable. For example, you can change the size and number of characters, the gap between characters, the colors of the “on” and “off” segments, and the size of the gap between segments.

Indicators architecture

The indicator classes consist of controls derived from an indicator base class and renderers derived from a character renderer base class. The renderer base class has two supporting classes at character level, one for data and one for the segment map that defines the character display.

The following UML diagrams show the relationships between the classes for indicator controls and the classes for indicator rendering respectively.





Using indicators

Using the NumericIndicator control

The `NumericIndicator` control is able to render digits and punctuation marks. It uses `NumericRenderer` (a 7-segment renderer) as the main character renderer and `ExtraCharRenderer` for punctuation marks.

The following example shows an indicator with 6 green numeric characters.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0x000000">
    <ilog:NumericIndicator width="100%" height="100"
      offSegmentColor="0x003300" onSegmentColor="0x00FF00"
      horizontalAlign="center" verticalAlign="middle"
      textHorizontalAlign="center"
      text="153.78" charHeight="50"/>
</mx:Application>
```

Using the AlphaNumericIndicator14 control

The `AlphaNumericIndicator14` control is able to render capital letters, digits and punctuation marks. It uses `AlphaNumericRenderer14` (a 14-segment renderer) as the main character renderer and `ExtraCharRenderer` for punctuation marks.

The following example shows an indicator with 15 green numeric characters.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"

                backgroundColor="0xFFFFFF">
    <ilog:AlphaNumericIndicator14 width="100%" height="100"
      offSegmentColor="0x003300" onSegmentColor="0x00FF00"
      horizontalAlign="center" verticalAlign="middle"
      textHorizontalAlign="center"
      text="IBM ILOG Elixir" charHeight="30" numChars="20"/>
</mx:Application>
```

Using the AlphaNumericIndicator35 control

The `AlphaNumericIndicator35` control is able to render uppercase and lowercase letters, digits, and punctuation marks. It uses `AlphaNumericRenderer35` (a 35-dot renderer) which can render any characters.

The following example shows an indicator with 15 green alphanumeric characters.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
```

```
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                backgroundColor="0xFFFFFF">
    <ilog:AlphaNumericIndicator35 width="100%" height="100"
      offSegmentColor="0x003300" onSegmentColor="0x00FF00"
      horizontalAlign="center" verticalAlign="middle"
      textHorizontalAlign="center"
      text="IBM ILOG Elixir" charHeight="30" numChars="20"/>
</mx:Application>
```

Styling your indicators

Changing the colors of indicator characters

You can change the “on” and “off” segment colors by changing the `onSegmentColor` and `offSegmentColor` style properties. Typically, `onSegmentColor` is much brighter than `offSegmentColor` to make the characters easily readable.

The default values are:

- `onSegmentColor`: `0xFF0000` (Light red)
- `offSegmentColor`: `0x330000` (Dark red)

Changing the height of indicator characters

Character height is controlled by the `charHeight` style property

In order to get enough room to render characters, `AlphaNumericRenderer35` (which corresponds to a 5x7 matrix) requires `charHeight` to be greater than or equal to 13 pixels (7 dots plus 6 gaps).

Changing the gap between indicator characters

You can change the gap between indicator characters using the `charGap` style property. The default is 2 pixels.

Changing the gap between character segments

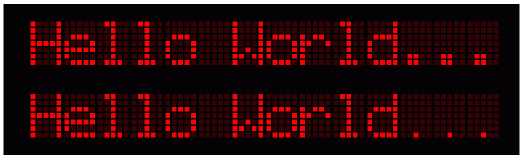
You can change the gap between segments of indicator characters using the `segmentGap` style property. Possible values are `none`, `thin`, `medium` and `large`.

Customizing the segment map

The following example shows how to customize the segment map that defines the character display for a dot.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical" xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
  backgroundColor="0x000000" backgroundGradientAlphas="[]">
  <mx:Script>
    <![CDATA[

      import ilog.indicators.indicatorsClasses.SegmentMap;
      import ilog.indicators.indicatorsClasses.AlphaNumericRenderer35;

      private function customizeMapping():void
      {
        // Creates a factory of renderers
        var cf:ClassFactory = new ClassFactory(AlphaNumericRenderer35);

        // Creates a copy of the default segment map
       var mySegmentMap:SegmentMap = AlphaNumericRenderer35.DEFAULT_MAP.clone
();

        // Changes the mapping for "."
        mySegmentMap.setOnSegments(".", [32]);
        cf.properties = {segmentMap: mySegmentMap};

        // Set the renderers factory
        indic2.renderer = cf;
      }
    ]]>
  </mx:Script>
  <ilog:AlphaNumericIndicator35 id="indic1" text="Hello World..."
charHeight="30" charGap="2"/>
  <ilog:AlphaNumericIndicator35 id="indic2" text="Hello World..."
charHeight="30" charGap="2"
    initialize="customizeMapping()"/>
</mx:Application>
```

The resulting difference in dot size from 4 elements to 1 element is shown in the following figure, with the “before” figure shown first.



Heat maps

Describes the use of IBM ILOG Elixir heat maps with Adobe® Flex® 3.

In this section

Introduction to heat maps
Describes the types of heat map available and gives an example of a heat map.

Heat map architecture
Describes the heat map classes.

Creating heat map controls
Describes how to create different types of heat map component.

Configuring heat maps
Describes how to configure different aspects of any heat map and different types of heat map.

Introduction to heat maps

An IBM ILOG Elixir heat map is a component that displays the graphical representation of a property for a set of points..

If the data points are *valued*, that is, each point is associated with a value, the graphical representation consists of regions known as *areas of same value* which can be filled with different colors or just be delineated by contour lines or both.

The following kinds of heat maps are available in IBM ILOG Elixir:

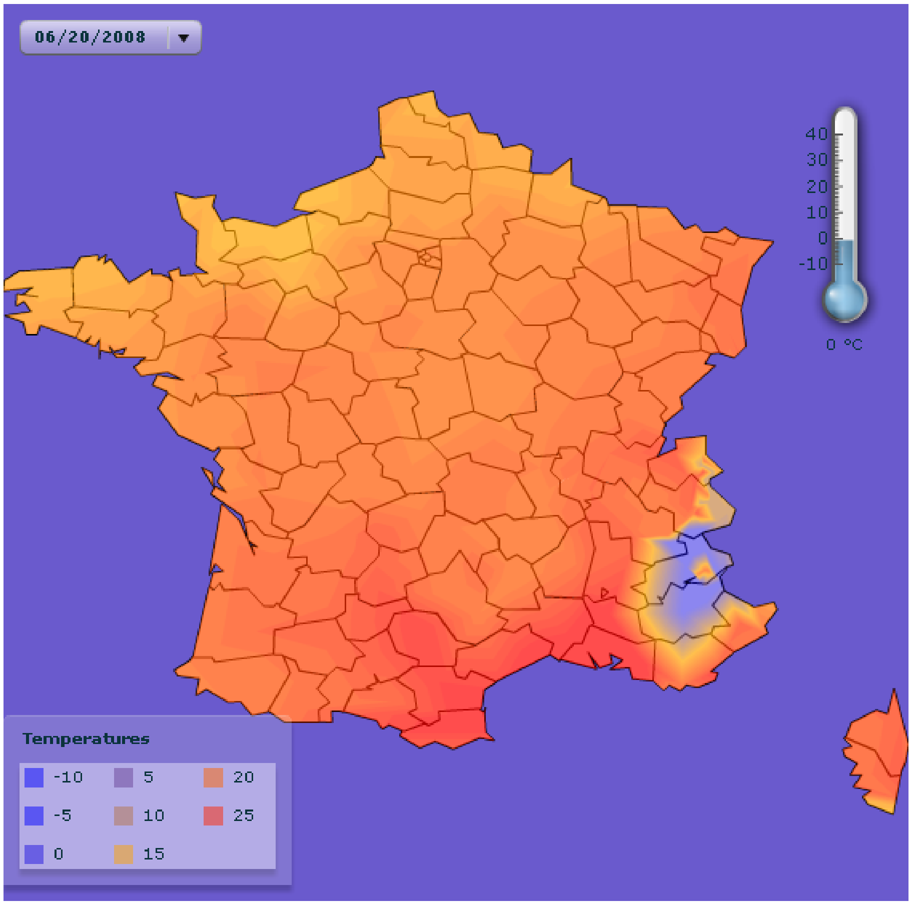
♦ The `ValuedHeatMap` control displays regions of same values for a set of valued points. These are drawn as *isometric contours* enclosing the *areas of same value* which can be filled with colors.

♦ The `DensityHeatMap` control is a density map drawn as a bitmap. Each point is drawn as a circle with a radial color gradient.

♦ The `MapHeatMap` control aggregates a heat map with a map control.

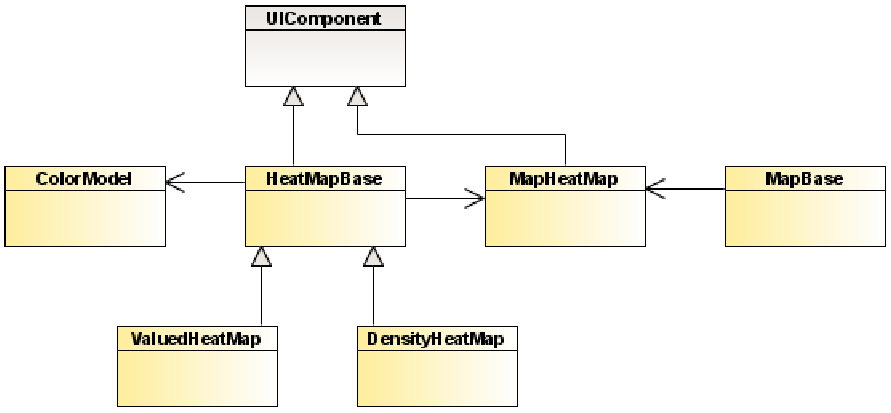
The data displayed by the heat maps is set through a `dataProvider` property. This property is converted to a collection of points, and the point set can be modified during conversion. Properties are available in the heat map controls to help in reading and converting input data.

The rendering of heat maps is controlled by a `ColorModel` object. The color model defines a set of colors associated with values, allowing you to retrieve a set of colors to represent a specified value. The colors returned are interpolated to match the value of the query. Properties specific to each heat map control are available to tune the display of the heatmap.


06/20/2008
40
30
20
10
0
-10
0 °C
Temperatures
-10
-5
0
5
10
15
20
25

Heat map architecture

The following figure shows the heat map classes.



The following table lists the heat map classes.

Heat map class	Description
HeatmapBase	The base class for heat maps.
ValuedHeatMap	Class used to display series of valued points.
DensityHeatMap	Class used to display density of points.
MapHeatMap	Class for aggregating a heat map and a map.

Creating heat map controls

Describes how to create different types of heat map component.

In this section

Creating a ValuedHeatMap component
Describes how to create a valued heat map.

Creating a DensityHeatMap component
Describes how to create a density heat map.

Creating a MapHeatMap component
Describes how to create a map heat map component comprising a map and a heat map.

Creating a ValuedHeatMap component

You can create a valued heat map control in MXML using the `<ilog:ValuedHeatMap>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value.

A heat map control displays the `value` property for a set of points using a `dataProvider` object as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <ilog:ValuedHeatMap width="100%" height="100%" >
    <ilog:dataProvider>
      <mx:XMLList>
        <point x = "173.2" y = "15.9" value = "60.1" />
        <point x = "97.3" y = "169.1" value = "72.5" />
        <point x = "297.5" y = "208.0" value = "69.4" />
        <point x = "386.4" y = "111.2" value = "32.0" />
        <point x = "326.7" y = "135.0" value = "64.0" />
        <point x = "85.2" y = "258.7" value = "50.0" />
        <point x = "358.0" y = "155.6" value = "50.0" />
        <point x = "305.0" y = "289.8" value = "41.1" />
        <point x = "199.3" y = "148.0" value = "90.0" />
        <point x = "208.0" y = "101.3" value = "86.1" />
        <point x = "48.7" y = "223.4" value = "44.8" />
        <point x = "10.7" y = "70.2" value = "22.4" />
        <point x = "234.9" y = "10.9" value = "57.1" />
        <point x = "367.1" y = "181.7" value = "43.7" />
        <point x = "339.9" y = "89.6" value = "52.8" />
        <point x = "361.7" y = "260.4" value = "28.6" />
        <point x = "236.2" y = "73.4" value = "78.5" />
        <point x = "98.8" y = "134.4" value = "73.2" />
        <point x = "373.0" y = "227.0" value = "32.6" />
        <point x = "279.4" y = "48.9" value = "63.6" />
        <point x = "0.0" y = "0.0" value = "0.0" />
        <point x = "400.0" y = "0.0" value = "0.0" />
        <point x = "400.0" y = "300.0" value = "0.0" />
        <point x = "0.0" y = "300.0" value = "0.0" />
      </mx:XMLList>
    </ilog:dataProvider>
    <ilog:colorModel>
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="100" alpha="1"/>
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:ValuedHeatMap>
</mx:Application>
```

The following figure shows the resulting valued heat map.

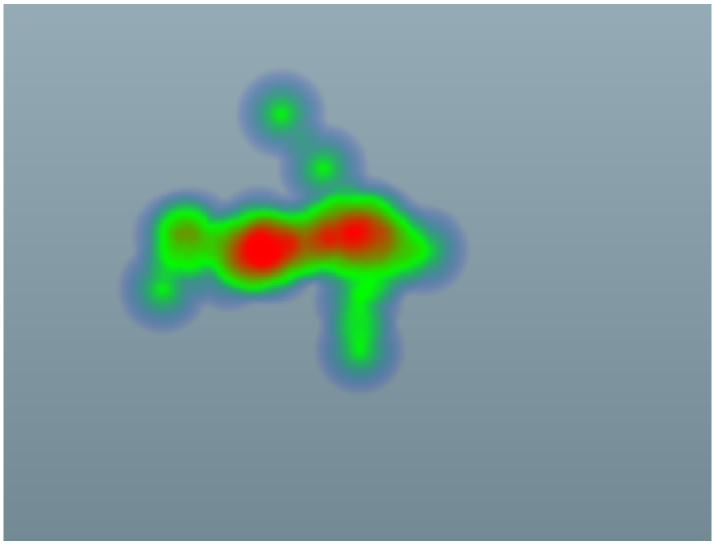
Creating a DensityHeatMap component

You can create a density heat map control in MXML using the `<ilog:DensityHeatMap>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value.

A heat map control displays the value property for a set of points using a `dataProvider` object as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <ilog:DensityHeatMap width="400" height="400">
    <ilog:dataProvider>
      <mx:XMLList>
        <point x = "179.8" y = "148.5" />
        <point x = "108.0" y = "144.1" />
        <point x = "221.2" y = "184.2" />
        <point x = "235.8" y = "141.5" />
        <point x = "198.0" y = "145.4" />
        <point x = "99.5" y = "177.6" />
        <point x = "225.3" y = "134.1" />
        <point x = "160.1" y = "158.7" />
        <point x = "118.3" y = "142.3" />
        <point x = "262.6" y = "152.9" />
        <point x = "173.4" y = "68.2" />
        <point x = "235.3" y = "159.1" />
        <point x = "169.6" y = "158.8" />
        <point x = "159.7" y = "140.9" />
        <point x = "222.7" y = "215.1" />
        <point x = "154.2" y = "150.5" />
        <point x = "203.9" y = "148.0" />
        <point x = "140.0" y = "163.6" />
        <point x = "216.0" y = "141.2" />
        <point x = "199.2" y = "102.3" />
      </mx:XMLList>
    </ilog:dataProvider>
    <ilog:colorModel >
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="120" />
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:DensityHeatMap>
</mx:Application>
```

The following figure shows the resulting density heat map.

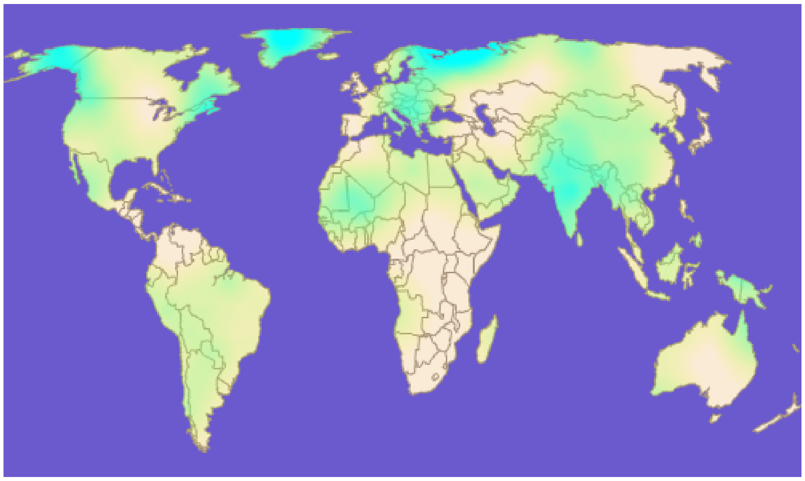


Creating a MapHeatMap component

The `MapHeatMap` control combines a heat map and a map control. It also provides the ability to clip the heat map to the map. The map and the heat map controls have to be set properly to get the `MapHeatMap` control to work correctly. The `MapHeatMap` control automatically sets the `dataFunction` property to the default data function for the heat map which assumes that the data provider provides objects with latitude/longitude coordinates. The map and heat map components are provided as properties of the `MapHeatMap` component as shown in the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
  <![CDATA[

    private function randomLongLat():Array {
      var ret:Array = [];
      for(var i:int = 0; i < 100; i++) {
        var o:Object = new Object();
        o.x = Math.random() * 360 - 180;
        o.y = Math.random() * 180 - 90;
        ret.push(o);
      }
      return ret;
    }

  ]]>
</mx:Script>
  <ilog:MapHeatMap width="400" height="200" clipToMap="true">
    <ilog:map>
      <ilog:WorldCountriesMap />
    </ilog:map>
    <ilog:heatMap>
      <ilog:DensityHeatMap id="heatMap"
                          dataProvider="{randomLongLat()}"
                          pointSize="15" >
        <ilog:colorModel>
          <ilog:ColorModel>
            <ilog:entries>
              <ilog:ColorEntry color="0xffff00" limit="0" alpha="0"/>
              <ilog:ColorEntry color="0x0fffff" limit="{heatMap.pointValue /
2}" alpha="0.5"/>
           </ilog:entries>
          </ilog:ColorModel>
        </ilog:colorModel>
      </ilog:DensityHeatMap>

    </ilog:heatMap>
  </ilog:MapHeatMap>
</mx:Application>
```

The following figure shows the resulting map of the world plus `DensityHeatMap` component filled randomly with latitude/longitude points.



Configuring heat maps

Describes how to configure different aspects of any heat map and different types of heat map.

In this section

Configuring heat map data
Describes how to configure the data on which a heat map is based.

Configuring a color model
Describes how to configure a color model used to render a heat map.

Configuring a ValuedHeatMap component
Describes how to configure the specific aspects of a valued heat map.

Configuring a DensityHeatMap component
Describes how to configure the specifc aspects of a density heat map

Configuring a MapHeatMap component
Describes how to configure the embedded map and heat map components in a `MapHeatMap` component and any extra options.

Adding a heat map legend
Describes how to add a legend to any type of heat map.

Configuring heat map data

Heat map data provider

A heat map component gets its data from a data provider, which contains a list of points to be displayed.

A heat map data provider can be one of the following types:

XML
: A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag or an `XML` or `XMLList` object.

IList or ICollectionView
: An object that implements the `IList` or `ICollectionView` interface, such as an instance of the `ArrayCollection` class or the `XMLListCollection` class, with a data provider that conforms to the structure specified in either of these items.

Other objects
: An array of items or an object.

When setting the `dataProvider` property the heat map control modifies its internal representation of the task data as follows:

- `XML` or `XMLList` are wrapped in an instance of `XMLListCollection`
- `IList` or `ICollectionView` are used directly.
- An `Array` object is wrapped in an instance of `ArrayCollection`
- An ActionScript® object of any other data type is wrapped in an `ArrayCollection` using an `Array` containing the object as its sole entry.

For example, you pass an `Array` to the data provider specified in the `dataProvider` property. If you read the data back from the data provider it is returned as an instance of `ArrayCollection`. The best practice when your event data can change dynamically is to use a collection, which provides the necessary notifications of changes.

Configuring the field mappings

By default, the heat map component looks for the `x`, `y`, and `value` fields of the objects provided by the data provider to get the coordinates and values of the point. You can change the fields to look for by setting the `xField`, `yField`, and `valueField` properties. These properties determine the name of the field in the data provider object that matches the `x`, `y`, or `value` field respectively.

In the following example, the fields of the data provider elements used for `x`, `y`, and `value` are `a`, `b`and `v` respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
```

```
<mx:Script>
  <![CDATA[
    [Bindable]
    private var dp:Array = [
      {a:15.6, b:227.8, v:25.9}, {a:30.1, b:153.9, v:43.8},
      {a:82.4,  b:272.0, v:44.0},{a:165.3, b:299.1, v:52.5},
      {a:192.8, b:104.3, v:86.6}, {a:131.2, b:137.3, v:82.2},
      {a:190.4, b:270.6, v:66.6},{a:253.9, b:17.1, v:57.1},
      {a:9.8,   b:79.3, v:24.1}, {a:330.9, b:290.9, v:30.8},
      {a:94.5,  b:174.7, v:71.2}, {a:193.7, b:242.3, v:76.3},
      {a:157.7, b:196.5, v:83.7}, {a:376.5, b:130.8, v:39.6},
      {a:333.7, b:184.0, v:59.6}, {a:277.7, b:76.0, v:71.6},
      {a:351.1, b:29.1, v:30.0},  {a:188.4, b:174.2, v:88.8},
      {a:379.5, b:208.9, v:32.9}, {a:70.4,  b:40.9, v:44.1}
    ];
  ]]>
</mx:Script>

  <ilog:ValuedHeatMap width="100%"
                      height="100%"
                      xField="a"
                      yField="b"
                      valueField="v"
                      dataProvider="{dp}">
    <ilog:colorModel >
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" />
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="100" alpha="1"/>
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:ValuedHeatMap>
</mx:Application>
```

Using a reference rectangle

If you want the heat map component to center your data points in the component automatically, set the `referenceRectangle` property. The component uses the position and size of the reference rectangle to determine which data points are included in the heat map. It translates and scales the coordinates of the data points in the reference rectangle to fit into the component. If a null rectangle is specified, the raw data points are displayed.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                 layout="absolute" xmlns:geom="flash.geom.*">
   <ilog:DensityHeatMap width="100%" height="100%" id="heatMap">
   <ilog:referenceRectangle>
     <geom:Rectangle x="120" y="110" width="130" height="120" />
   </ilog:referenceRectangle>
     <ilog:dataProvider>
       <mx:XMLList>
         <point x="179.8" y="148.5" /><point x="108.0" y="144.1" />
```

```
        <point x="221.2" y="184.2" /><point x="235.8" y="141.5" />
        <point x="198.0" y="145.4" /><point x="99.5" y="177.6" />
        <point x="225.3" y="134.1" /><point x="160.1" y="158.7" />
        <point x="118.3" y="142.3" /><point x="262.6" y="152.9" />
        <point x="173.4" y="68.2" /> <point x="235.3" y="159.1" />
        <point x="169.6" y="158.8" /><point x="159.7" y="140.9" />
        <point x="222.7" y="215.1" /><point x="154.2" y="150.5" />
        <point x="203.9" y="148.0" /><point x="140.0" y="163.6" />
        <point x="216.0" y="141.2" /><point x="199.2" y="102.3" />
      </mx:XMLList>
    </ilog:dataProvider>
    <ilog:colorModel >
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="120" />
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:DensityHeatMap>
</mx:Application>
```

Using a data function

You can set a data function to transform the coordinates and values found in the data provider yourself by setting the `dataFunction` property. The data function must have the following signature: `dataFunction(o:Object):Object`. An original object is passed to the data function and the caller of the data function expects a new object to be returned.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                 layout="absolute">

   <mx:Script>
     <![CDATA[
       private function dataF(o:XML):Object {
         var x:Number = o["@x"];
         var y:Number = o["@y"];
         var ret:XML = o.copy();
         ret["@x"] = 50 + x * width / 500;
         ret["@y"] = 50 + y * height / 500;
         return ret;
       }
     ]]>
   </mx:Script>

   <ilog:DensityHeatMap width="100%" height="100%" id="heatMap"
     dataFunction="dataF" >
     <ilog:dataProvider>
       <mx:XMLList>
         <point x="179.8" y="148.5" /><point x="108.0" y="144.1" />
         <point x="221.2" y="184.2" /><point x="235.8" y="141.5" />
         <point x="198.0" y="145.4" /><point x="99.5" y="177.6" />
         <point x="225.3" y="134.1" /><point x="160.1" y="158.7" />
         <point x="118.3" y="142.3" /><point x="262.6" y="152.9" />
         <point x="173.4" y="68.2" /> <point x="235.3" y="159.1" />
         <point x="169.6" y="158.8" /><point x="159.7" y="140.9" />
         <point x="222.7" y="215.1" /><point x="154.2" y="150.5" />
         <point x="203.9" y="148.0" /><point x="140.0" y="163.6" />
         <point x="216.0" y="141.2" /><point x="199.2" y="102.3" />
       </mx:XMLList>
     </ilog:dataProvider>
     <ilog:colorModel >
      <ilog:ColorModel>
       <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
       <ilog:ColorEntry color="0x00ff00" limit="50" />
       <ilog:ColorEntry color="0xff0000" limit="120" />
      </ilog:ColorModel>
     </ilog:colorModel>
   </ilog:DensityHeatMap>
</mx:Application>
```

Configuring a color model

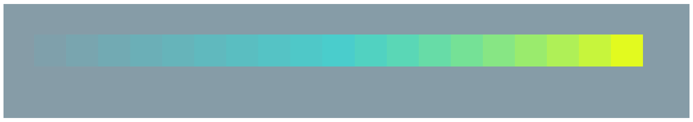
The `colorModel` property of a heat map component allows you to specify a color model to map values to colors. You can configure the color model by setting up an array of `colorEntry` objects. A color entry has three properties: a `color` property, a `limit` property, and an `alpha` property. The `color` property defines the color for this entry. The `limit` property is the value at which the color is bound. The `alpha` property defines the transparency of the color for this entry. Note that the color entries have to be specified in strict descending order with respect to the `limit` property.

A color model is used in each heat map component to provide visual rendering. Once the color model is configured, it is possible to query colors for a given value. The querying of colors is done automatically in each component so that you do not have to query colors manually. Colors corresponding to values that do not match a color entry are interpolated from the colors found at adjacent entries.

The following code sample draws squares whose colors are taken from a color model.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
  <![CDATA[
  public override function validateDisplayList():void {
    var g:Graphics = graphics;
    var size:int = 20;
    for(var i:int = 0; i < size; i++) {
      var value:Number = size * i;
      g.beginFill(colors.getColor(value), colors.getAlpha(value));
      g.drawRect(size * i, size, size, size);
      g.endFill();
    }
  }
  ]]>
</mx:Script>

  <ilog:ColorModel id="colors">
    <ilog:ColorEntry color="0x00ffff" alpha="0" limit="1" />
    <ilog:ColorEntry color="0x0ffff0" alpha="0.5" limit="200" />
    <ilog:ColorEntry color="0xffff00" alpha="1" limit="400" />
  </ilog:ColorModel>

</mx:Application>
```

The following figure shows the resulting color squares.



Configuring a ValuedHeatMap component

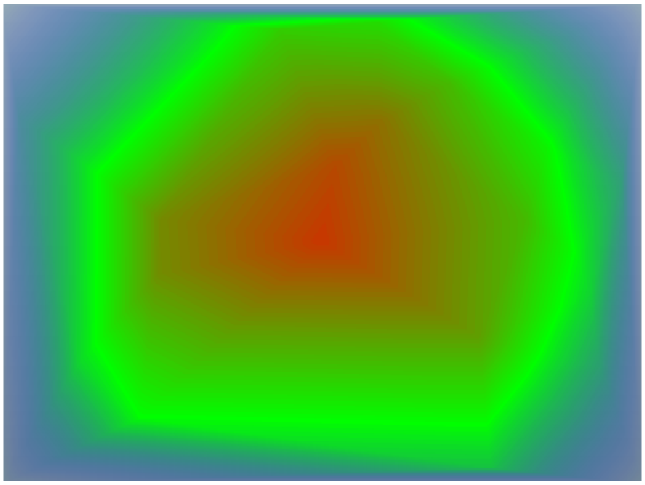
ValuedHeatMap parameters

The `ValuedHeatMap` component takes three parameters from the objects provided by the data provider: the `x` location of the point, the `y` location of the point, and the `value` of the point. These parameters are used to construct the isometric contour of the set of points using the `values` property. The `values` property will be used to determine the value at which an isometric contour will be drawn. The more values you provide in the `values` property, the smoother the drawing, but the heavier the display time and memory consumption.

The following example computes an array of values giving more precise and smooth colors. The values are computed using a function to avoid a long enumeration in MXML code.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
<mx:Script>
  <![CDATA[
     private function computeValues():Array {
       var ret:Array = [];
       for(var i:int = 0; i < 100; i++)
        ret.push(i);
       return ret;
     }
  ]]>
</mx:Script>
  <ilog:ValuedHeatMap width="100%" height="100%" values="{computeValues()}">
    <ilog:dataProvider>
      <mx:XMLList>
        <point x = "15.6" y = "227.8" value = "25.9" />
        <point x = "30.1" y = "153.9" value = "43.8" />
        <point x = "82.4" y = "272.0" value = "44.0" />
        <point x = "165.3" y = "299.1" value = "52.5" />
        <point x = "192.8" y = "104.3" value = "86.6" />
        <point x = "131.2" y = "137.3" value = "82.2" />
        <point x = "190.4" y = "270.6" value = "66.6" />
        <point x = "253.9" y = "17.1" value = "57.1" />
        <point x = "9.8" y = "79.3" value = "24.1" />
        <point x = "330.9" y = "290.9" value = "30.8" />
        <point x = "94.5" y = "174.7" value = "71.2" />
        <point x = "193.7" y = "242.3" value = "76.3" />
        <point x = "157.7" y = "196.5" value = "83.7" />
        <point x = "376.5" y = "130.8" value = "39.6" />
        <point x = "333.7" y = "184.0" value = "59.6" />
        <point x = "277.7" y = "76.0" value = "71.6" />
        <point x = "351.1" y = "29.1" value = "30.0" />
        <point x = "188.4" y = "174.2" value = "88.8" />
        <point x = "379.5" y = "208.9" value = "32.9" />
        <point x = "70.4" y = "40.9" value = "44.1" />
      </mx:XMLList>
    </ilog:dataProvider>
    <ilog:colorModel>
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="100" alpha="1"/>
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:ValuedHeatMap>
</mx:Application>
```

The following figure shows the resulting smoother heatmap.



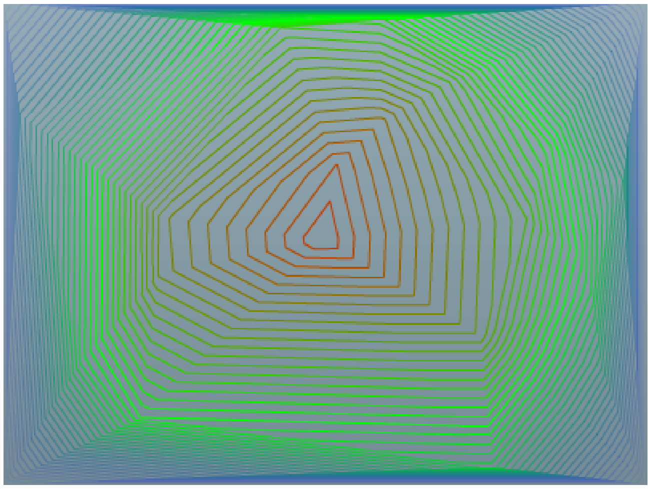
ValuedHeatMap rendering options

A valued heat map allows you to vary the following rendering options:

showFill
Set `showFill` to `true` to have the same-value areas filled.

showStroke
Set `showStroke` to true to have the isometric contour lines drawn.

You can set these two properties independently to have only area fill, or only contour lines (as in the following figure), or both.



Configuring a DensityHeatMap component

DensityHeatMap parameters

The `DensityHeatMap` component draws a heat map that displays information about the density of data points. It uses a `BitmapData` object to draw this information directly. Each data point in the data provider is drawn as a radial gradient in the pixmap, providing a value between `0` and `pointValue` for each pixel. The blending of data points with other points is controlled by the `pointBlendMode` property.

DensityHeatMap rendering options

A density heat map allows you to vary the following rendering options:

doubleBuffer
When this property is set to `true`, the `DensityHeatMap` object uses two buffers for the final rendering. One buffer is used to accumulate values to be rendered and a second buffer is used to build the final image by translating values to colors. Double buffering gives faster palette switching and faster dynamic point insertion. However, as this mode uses two buffers, there is higher memory consumption.

pointValue
This property controls the value of a data point drawn in the bitmap. This is the value corresponding to the center of the circle representing the data point.

pointSize
This property defines the size of a data point. Each data point is drawn as a circular gradient whose radius is `pointSize`.

pointBlendMode
A string value, from the `flash.display.BlendMode` class, specifying the blend mode to be applied to the resulting bitmap when drawing points adjacent to the data point.

pixelValueFunction
This property defines a function that is used to return the value corresponding to a given pixel. This function can be defined by the user to fit the application needs. A default implementation is provided in the `DensityHeatMap` control which returns the grey value of the pixel.

Palette
The palette property is an array of 256 colors that can be provided to do the final rendering of the density heatmap. At rendering time, each pixel value is converted to a color using this palette. If no palette is provided but a color model is provided through the `colorModel` property, a palette is derived from this color model.

Configuring a MapHeatMap component

To configure the `MapHeatMap` component, you have to configure the embedded map and heat map components.

MapHeatMap parameters

The parameters available depend on the type of heat map. See *ValuedHeatMap parameters* or *DensityHeatMap parameters*.

MapHeatMap rendering options

A map heat map allows you to vary the following extra rendering options:

clipToMap
: To clip the heat map, set the `clipToMap` property. When using a `MapHeatMap` component, you can clip the drawing of the heat map precisely to the map using the `clipToMap` property. When this property is set to `true`, the heat map is clipped to the map contour.

resample
: To get more precise rendering when using a `DensityHeatMap` component with the `MapHeatMap` component, set the `resample` property. This property recomputes the bitmap instead of just resizing it after a zoom operation. As recomputing the bitmap is more time-consuming than resizing it, setting this property to `true` will slow down the display of the component.

The following code sample shows a `MapHeatMap` component containing a `WorldCountriesMap` component with a randomly filled `DensityHeatMap` component. The `clipToMap` and `resample` properties are controlled by two check boxes.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 layout="absolute"
                 xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                 xmlns:local="*" >
<mx:Script>
   <![CDATA[
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     import ilog.maps.MapBase;

     private var _numPoints:int = 300;
     private var _points:ArrayCollection = null;

     private function randRange(min:Number, max:Number):Number {
       var r:Number = Math.random();
       return r * (max - min) + min;
     }

     private function randomLongLat():ArrayCollection {
       if(_points == null) {
         var ret:Array = [];
         for(var i:int = 0; i < _numPoints; i++) {
           var o:Object = new Object();
           o.x = randRange(-180, 180);
           o.y = randRange(-90, 90);
           ret.push(o);
         }
         _points = new ArrayCollection(ret);
       }
       return _points;
     }

     private function convert(o:Object):Object {
       var lon:Number = o.x;
       var lat:Number = o.y;
       var p:Point = new Point(o.x, o.y);
       p = map.latLongToCanvas(p);
       p = map.drawingCanvas.localToGlobal(p);
       p = heat.globalToLocal(p);
       o.x = p.x;
       o.y = p.y;
       return o;
     }

     private function click(e:Event):void {
       heat.dataProvider = randomLongLat();
     }

   ]]>
</mx:Script>
<mx:Canvas width="100%" height="100%">

   <ilog:MapHeatMap width="100%" height="100%"
                    clipToMap="{clip.selected}"
                    id="mapheatmap"
                    resample="{resample.selected}">
     <ilog:map>
       <local:World_countriesMap id="map"
                                 allowNavigation="true"
                                 fill="0x21177D"
                                 stroke="0x008E8E"
                                 backgroundFill="{new
SolidColor(0xCFCCED, 0)}" >
         <local:filters>
           <mx:DropShadowFilter distance="10" color="0"/>
         </local:filters>
       </local:World_countriesMap>
     </ilog:map>
     <ilog:heatMap>
       <ilog:DensityHeatMap pointValue="200"
                            id="heat"
                            mouseEnabled="false"
                            dataProvider="{randomLongLat()}"
                            pointSize="25">
         <ilog:colorModel>
```

```
           <ilog:ColorModel>
             <ilog:entries>
               <ilog:ColorEntry color="0x000fff" limit="0" alpha="0"/>
               <ilog:ColorEntry color="0x00fff0" limit="{255 / 3}"
alpha="0.3"/>
               <ilog:ColorEntry color="0x0fff00" limit="{2 * 255 / 3}"
alpha="0.6" />
               <ilog:ColorEntry color="0xfff000" limit="255" alpha="1"/>
             </ilog:entries>
           </ilog:ColorModel>
         </ilog:colorModel>
       </ilog:DensityHeatMap>

     </ilog:heatMap>
   </ilog:MapHeatMap>
   <mx:CheckBox id="clip" label="Clip" left="20" bottom="5" />
   <mx:CheckBox id="resample" label="Resample" left="100" bottom="5" />
</mx:Canvas>
</mx:Application>
```

Adding a heat map legend

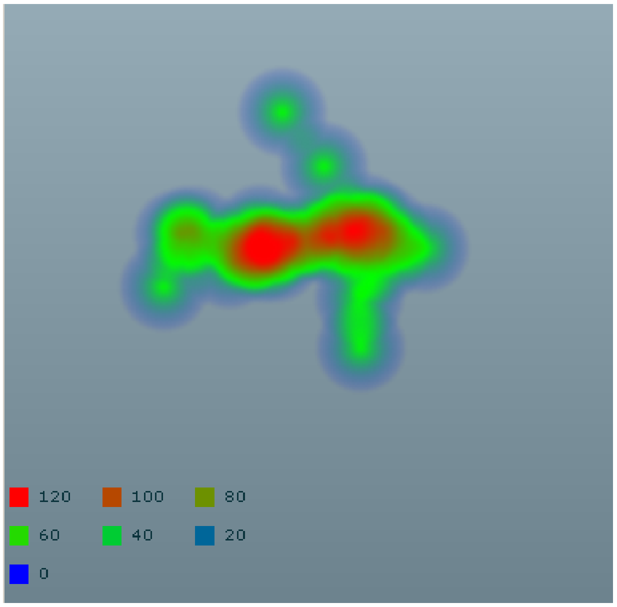
A legend can be connected to any heat map component. It can be filled automatically if the heat map control has a `colorModel` property and a `values` property. In this case, a legend item will be created for each value found in the `values` property with the corresponding color found in the color model.

To connect the `HeatMapLegend` component to the heat map control, set the heat map control as the `dataProvider` property of the legend. Note that you can also provide an array of `LegendItem` objects as the data provider of the legend if you prefer building your legend items by hand.

The following example shows how to attach a legend to a `DensityHeatMap` component that has a color model.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute">
  <ilog:DensityHeatMap width="400" height="300" id="heatMap" >
    <ilog:dataProvider>
      <mx:XMLList>
        <point x="179.8" y="148.5" /><point x="108.0" y="144.1" />
        <point x="221.2" y="184.2" /><point x="235.8" y="141.5" />
        <point x="198.0" y="145.4" /><point x="99.5" y="177.6" />
        <point x="225.3" y="134.1" /><point x="160.1" y="158.7" />
        <point x="118.3" y="142.3" /><point x="262.6" y="152.9" />
        <point x="173.4" y="68.2" /> <point x="235.3" y="159.1" />
        <point x="169.6" y="158.8" /><point x="159.7" y="140.9" />
        <point x="222.7" y="215.1" /><point x="154.2" y="150.5" />
        <point x="203.9" y="148.0" /><point x="140.0" y="163.6" />
        <point x="216.0" y="141.2" /><point x="199.2" y="102.3" />
      </mx:XMLList>
    </ilog:dataProvider>
    <ilog:colorModel >
     <ilog:ColorModel>
      <ilog:ColorEntry color="0x0000ff" limit="0" alpha="0"/>
      <ilog:ColorEntry color="0x00ff00" limit="50" />
      <ilog:ColorEntry color="0xff0000" limit="120" />
     </ilog:ColorModel>
    </ilog:colorModel>
  </ilog:DensityHeatMap>
  <ilog:HeatMapLegend y="{heatMap.y + heatMap.height}" dataProvider="{heatMap}
" >
    <ilog:values>
      <mx:Number>120</mx:Number>
      <mx:Number>100</mx:Number>
      <mx:Number>80</mx:Number>
      <mx:Number>60</mx:Number>
      <mx:Number>40</mx:Number>
      <mx:Number>20</mx:Number>
      <mx:Number>0</mx:Number>
    </ilog:values>
  </ilog:HeatMapLegend>
</mx:Application>
```

The following figure shows the resulting `DensityHeatMap` object with legend.



Maps

Describes the use of IBM ILOG Elixir maps with Adobe® Flex® 3.

In this section

Introduction to maps
Describes the uses and features of maps with an example.

Map architecture
Describes the architecture of the map classes.

Map features
Explains what map features are and how to use them.

Creating a map control
Describes how to create a map control in two different ways.

Styling maps
Describes the ways of customizing a map.

Managing map events
Describes the management of events on maps.

User interaction with maps
Describes the ways that you can interact with a map.

Using effects in a map
Describes the use of map effects.

Using built-in effects in a map
Describes the two built-in effects: color transition effect and fit effect.

Using extra map features
Describes the following extra map features: drawing, longitude/latitude API.

Generating custom maps
Describes how to use the IBM® ILOG® Elixir Custom Map Converter tool for generating custom maps from Shapefiles.

Introduction to maps

IBM ILOG Elixir provides map display capabilities for creating map-based dashboards. The components can be used to display information such as sales revenues or stock levels in all the countries that you operate in.

Each map item, such as a country or a state, can be panned or zoomed, is selectable, and has a highlight color to indicate that it is selected.

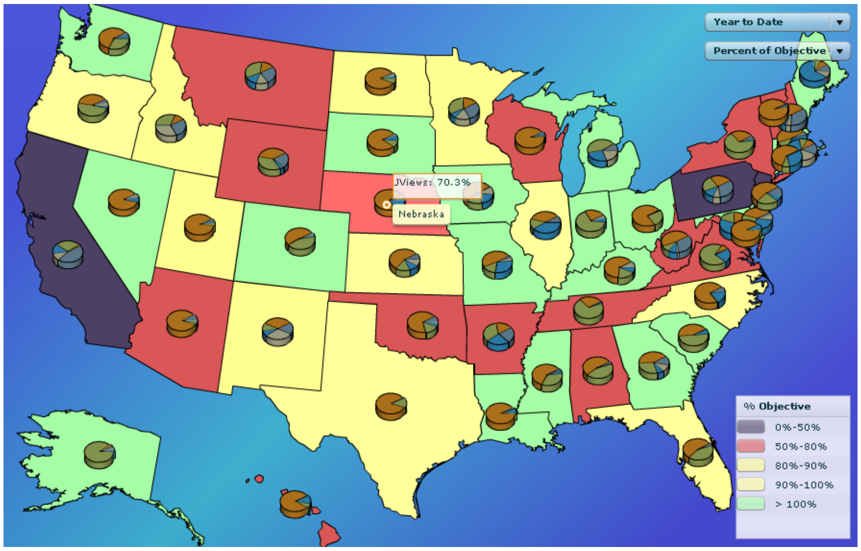
You can overlay your color-coded map display with any Adobe® Flex® objects such as custom charts, buttons, grids, and icons.

There are six maps provided in IBM ILOG Elixir as subclasses of the `MapBase` class:

- A world map showing all countries: `WorldCountriesMap`
- A map of the United States showing all the states: `USStatesMap`
- A map of Continental Europe: `ContinentalEuropeMap`
- A map of the Asia/Pacific region: `AsiaPacificMap`
- A map of the Americas: `AmericasMap`
- A map of Europe, the Middle East, and Africa: `EuropeMiddleEastAfricaMap`

The following map shows a number of features that you can program into your map display.

For example, states are highlighted and a tool tip is displayed when the pointer is rolled over them, and each state has a pie chart showing the product sales split.

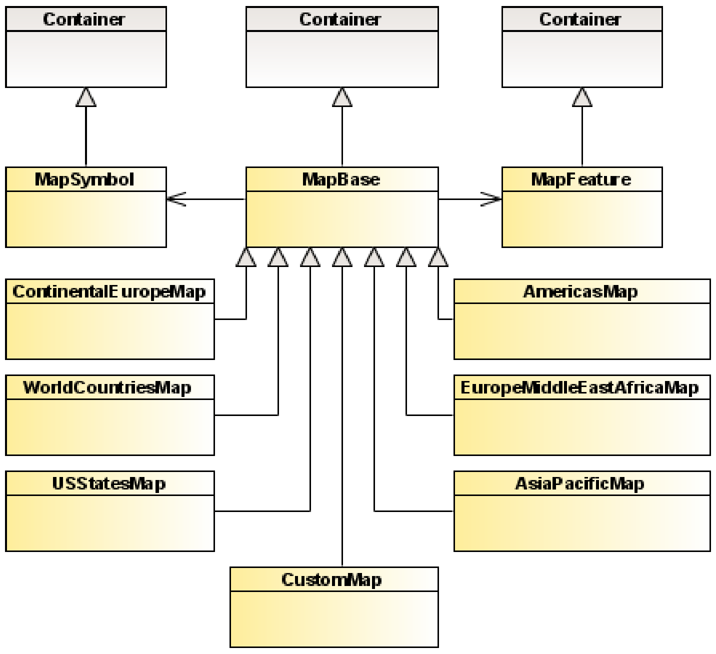


Map architecture

A map control is a subclass of the `MapBase` class. It displays a set of map items called map features, which are instances of the `MapFeature` class.

Symbols on maps are instances of the `MapSymbol` class.

A map control can be a predefined component supplied with the product or a custom map generated with the IBM® ILOG® Elixir Custom Map Converter.



Map features

A map feature is a subclass of a container representing, for example, a country in Europe or Asia or a state in the case of the United States map. Each map feature is identified by its key, which is the country code or state code (using the ISO 3166-1 alpha-3, three-letter system, or the two-letter system for the United States). The corresponding class is the `MapFeature` class.

By default, a `MapFeature` instance is created for each country or state in a map. You can retrieve this instance by using the `getFeature()` method of the `MapBase` class.

If you need to create your own `MapFeature` instance, you can add it as a child of a map by using either the API or MXML. This instance replaces the default instance with the same key.

Once you have a `MapFeature` instance, you can:

- ♦ Style it using the various coloring properties of the `MapFeature` object.
- ♦ Place symbols on it located at a predefined position to provide additional information about the state or country. A symbol is an instance of the `MapSymbol` class, which is a subclass of `UIComponent`.

You can group features of the map. The result is a new `MapFeature` object containing the grouped features.

There are two ways to group features:

- ♦ Call the method `groupFeatures` in ActionScript®
- ♦ Declare a new feature in the MXML file

To ungroup previously grouped features, call the method `ungroupFeature`.

The following example makes two groups of European countries in MXML and shows how to ungroup one of them in ActionScript. In this example, the user can click a country within a group to ungroup the countries.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 layout="absolute"
                 xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
xmlns:maps="ilog.maps.*">
<mx:Script>
   <![CDATA[
     import ilog.maps.MapBase;
     import ilog.maps.MapEvent;
     import ilog.maps.MapFeature;
     import ilog.maps.Map;

     private function itemClick(e:MapEvent):void {
       var mf:MapFeature = e.mapFeature;
       if(mf != null && mf.key == "Group1")
         MapBase(e.currentTarget).ungroupFeature(mf);
     }
   ]]>
</mx:Script>
   <ilog:ContinentalEuropeMap allowNavigation="true"
                              width="500"
                              height="500"
                              x="50"
                              y="50"
                              mapItemClick="itemClick(event)">

    <ilog:MapFeature key="Group1" fill="0xff0000">
      <ilog:features>
        <mx:String>CHE</mx:String>
        <mx:String>GBR</mx:String>
     </ilog:features>
    </ilog:MapFeature>

    <ilog:MapFeature key="Group2" fill="0xffff00">
      <ilog:features>
        <mx:String>GRC</mx:String>
        <mx:String>FRA</mx:String>
        <mx:String>ITA</mx:String>
     </ilog:features>
    </ilog:MapFeature>

   </ilog:ContinentalEuropeMap>
</mx:Application>
```

Creating a map control

You can create a map control in two ways:

- ♦ You can create, destroy, and manipulate map displays using ActionScript® as for any other IBM ILOG Elixir component.
- ♦ You can create a map in MXML using the tag corresponding to the map you want to create.

To create a map of Europe in ActionScript:

1. Use the `new` keyword.
2. Set properties for the map object in the same way as you can in MXML.
3. Color a map item as a country by retrieving its map feature and then setting the appropriate style property.
4. Make a map react to user actions by using an event listener.

The following example displays a map of Europe. France is colored in green and the country code is displayed when the user clicks on the country.

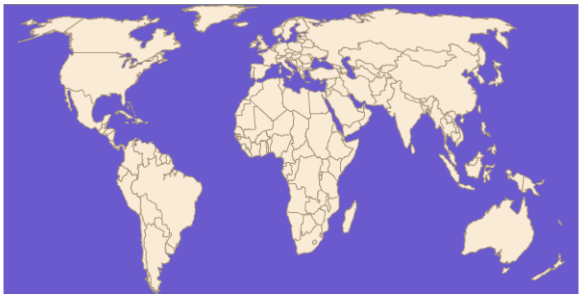
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="creationComplete()">

<mx:Script>
  <![CDATA[
    import mx.graphics.SolidColor;
    import ilog.maps.MapBase;
    import ilog.maps.MapEvent;
    import ilog.maps.MapFeature;
    import ilog.maps.ContinentalEuropeMap;

    internal function creationComplete():void {

      // Create a map of Continental Europe.
      var map:ContinentalEuropeMap = new ContinentalEuropeMap();
      map.width = 200;
      map.height = 200;
      panel.addChild(map);

      // Color France in green.
      var mf:MapFeature = map.getFeature("FRA");
      mf.setStyle(MapBase.FILL, new SolidColor(0x00ff00));

      // Add an event listener to display the country code of the country.
      // Clicked.
      map.addEventListener(MapEvent.ITEM_CLICK, itemClicked);
    }

    private function itemClicked(e:MapEvent):void {
      var mf:MapFeature = e.mapFeature;
      if(mf != null)
        trace(mf.key + " clicked");
    }
]]>

</mx:Script>
<mx:Panel id="panel" width="100%" height="100%"
    title="Map created in ActionScript" />
</mx:Application>
```

To create a map of the countries of the world using MXML:

1. Use the `<ilog:WorldCountriesMap>` tag.
2. Insert this tag in an Application element as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"

                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:WorldCountriesMap width="300" height="150" />
</mx:Application>
```

The following figure shows the map displayed by the code above.



Styling maps

Describes the ways of customizing a map.

In this section

Overview of map customization
Describes the facilities for customizing a map.

Using CSS
Describes the use of CSS for styling a map control.

Using inline properties
Describes the use of inline properties for styling the map background.

Using tags and tag bindings
Describes the use of MXML tags and tag bindings for styling a map.

Using symbols
Describes the use of symbols to enhance a map.

Using a repeater control
Describes the use of a repeater control to apply the same styling to all data items.

Overview of map customization

You can customize your map display by changing the visual aspect of the map, for example, by assigning different colors for each country. Customization includes the ability to change the `fill` and `stroke` style properties of a country, the `fill` and `stroke` properties of a country when it is highlighted, and the filters that can be applied to each country or state. The background of the map can also be customized.

Using CSS

You can use the control name in CSS to define styles for a control. This is referred to as a type selector, because the style you define is applied to all controls of that type.

For example, the following style definition specifies the fill color and highlight fill color for all the `ContinentalEuropeMap` controls. In addition, a special `stroke` is defined for Ireland using a class selector.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                layout="absolute" >

<mx:Style>
  .irl {
    stroke : #ff0000;
  }

  ContinentalEuropeMap {
    fill : #333333;
    highlight-fill : #cecece;
  }
</mx:Style>

<ilog:ContinentalEuropeMap
  id="europe"
  x="10"
  y="20"
  width="600"
  height="400"
  allowNavigation="true"
  zoomableSymbols="true">

  <ilog:MapFeature key="IRL" styleName="irl" />

</ilog:ContinentalEuropeMap>

</mx:Application>
```

Using inline properties

To change the background of a map, use the `backgroundFill` tag.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">

<ilog:ContinentalEuropeMap id="europe" width="100%" height="100%">
<!-- Change the map background to a gradient -->
  <ilog:backgroundFill>
    <mx:LinearGradient angle="33">
      <mx:entries>
        <mx:GradientEntry color="0x0000ff" ratio="0" alpha=".5"/>
        <mx:GradientEntry color="0x00ccff" ratio=".5" alpha=".5"/>
        <mx:GradientEntry color="0x0f00ff" ratio="1" alpha=".5"/>
      </mx:entries>
    </mx:LinearGradient>
  </ilog:backgroundFill>
</ilog:ContinentalEuropeMap>
</mx:Application>
```

Using tags and tag bindings

You can define styles with MXML tags and then bind the values of the renderer properties to those tags. The following example defines a gradient fill for all the countries of Europe and sets a special line weight for the `stroke` style property for Ireland.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Stroke color="0xff0000" weight="3" id="irl" />
<mx:LinearGradient id="fill" >
<mx:GradientEntry color="0x0000ff" ratio="0" />
<mx:GradientEntry color="0x000000" ratio="1" />
</mx:LinearGradient>

<mx:Stroke color="0xff0000" weight="3" id="irl" />
<mx:LinearGradient id="fill" >
  <mx:GradientEntry color="0x0000ff" ratio="0" />
  <mx:GradientEntry color="0x000000" ratio="1" />
</mx:LinearGradient>

<ilog:ContinentalEuropeMap

id="europe"
x="10" y="20" width="600" height="400"
fill="{fill}"
allowNavigation="true" >
  <!-- A thick red line for Ireland -->
  <ilog:MapFeature key="IRL" stroke="{irl}" />
</ilog:ContinentalEuropeMap>
</mx:Application>
```

Using symbols

Maps can be enhanced by the use of symbols. Symbols are `UIComponent`s that are placed on top of the map to add complementary information about a map item. When a map is zoomed, symbols can be zoomed with it (or not), depending on the value of the `zoomableSymbols` property of the map control. The following example adds a pie chart on top of Spain:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
<![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
    {Expense:"Taxes", Amount:2000},
    {Expense:"Rent", Amount:1000},
    {Expense:"Bills", Amount:100},
    {Expense:"Car", Amount:450},
    {Expense:"Gas", Amount:100},
    {Expense:"Food", Amount:200}
  ]);
]]>
</mx:Script>

<ilog:ContinentalEuropeMap id="europe" width="100%" height="100%">
  <!-- Add a PieChart on top of Spain, key is 'ESP' -->
  <ilog:MapSymbol key="ESP" >
    <mx:PieChart dataProvider="{expenses}" showDataTips="true"
      width="100" height="100">
      <mx:series>
        <mx:PieSeries field="Amount"
          nameField="Expense" labelPosition="callout" />
      </mx:series>
    </mx:PieChart>
  </ilog:MapSymbol>
</ilog:ContinentalEuropeMap>

</mx:Application>
```

By default, symbols are placed using a reference point predefined for each country or state. The symbols are then positioned on the map at the reference point of the map feature corresponding to the key value of the symbol.

An alternative placement is possible by using latitude and longitude values in decimal degrees. If a symbol does not provide a key value, but provides latitude and longitude values, these coordinates are used to place the symbol on the map.

The following example shows how to place a symbol at San Fransisco, longitude -122.5 degrees and latitude 37.5 degrees:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<ilog:USStatesMap id="map"
                  width="100%"
                  height="100%">
   <ilog:MapSymbol longitude="-122.5" latitude="37.5">
    <mx:Button width="20" height="20" />
   </ilog:MapSymbol>
</ilog:USStatesMap>
</mx:Application>
```

Using a repeater control

If you want to iterate the same kind of styling over map features using data from a data provider, you can use a `Repeater` object as the single child of a map. The `dataProvider` property of the repeater points to your data provider. The repeater takes its `MapFeature` child and instantiates it for each data item in the `dataProvider`. For more details about the Adobe® Flex® `Repeater` object, see *Using the Repeater* in the *Adobe Flex 3.0 Developer's Guide*.

In the following example, the `dataProvider` is an `ArrayCollection` that contains all the styles of the map in a single array. Writing the map customization in this way is more concise than writing the equivalent customization without the repeater.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
   <![CDATA[
     import mx.collections.HierarchicalCollectionView;
     import mx.effects.Blur;
     import mx.graphics.GradientEntry;
     import mx.graphics.LinearGradient;
     import mx.collections.ArrayCollection;
   internal static var gradient:LinearGradient = new LinearGradient();
   gradient.entries = [
     new GradientEntry(0xff0000, 0, 1),
     new GradientEntry(0x00ff00, 0.5, 1),
     new GradientEntry(0x0000ff, 1, 1)
   ];
   internal static var blur:BlurFilter = new BlurFilter();
   [Bindable]
   private var countries:ArrayCollection = new ArrayCollection([
     { country : "GBR", color : "green", line : "red" ,
      highlightFill : gradient, highlightStroke : 0xff0000 },
     { country : "FRA", color : "blue", line : "yellow" },
     { country : "ITA", filters : [blur] }
   ]);
    ]]>
</mx:Script>
<ilog:ContinentalEuropeMap
   id="europe"
   x="10"
   y="20"
   width="600"
   height="400">
   <mx:Repeater dataProvider="{countries}" id = "r">
     <ilog:MapFeature key = "{r.currentItem.country}"
                      fill = "{r.currentItem.color}"
                      stroke = "{r.currentItem.line}"
                      highlightFill= "{r.currentItem.highlightFill}"
                      highlightStroke = "{r.currentItem.highlightStroke}"
                      filters="{r.currentItem.filters}"
                      highlightFilters="{r.currentItem.hfilters}" >
     </ilog:MapFeature>
   </mx:Repeater>
</ilog:ContinentalEuropeMap>
</mx:Application>
```

Managing map events

Map controls accept several kinds of user interaction, from moving the pointer over a data point to clicking or double-clicking a map item as a country or a state. You can create an event handler for each of these interactions and use the `MapEvent` class in this handler to access the data related to this interaction.

Map data events that you can listen for are shown in the following table.

Map data events

Map data event	Description
`mapChange`	Indicates that the selection changed as a result of user interaction.
`mapItemClick`	Indicates that the user clicked the pointer over a visual item in the control.
`mapItemDoubleClick`	Indicates that the user double-clicked the pointer over a visual item in the control.
`mapItemRollOut`	Indicates that the user rolled the pointer out of a visual item in the control.
`mapItemRollOver`	Indicates that the user rolled the pointer over a visual item in the control.

The map data events are of the type `MapEvent`. In the following example, the map features are selected when the pointer enters a country area. Rotate the mouse wheel forward or backward to fit the selected country or the entire map onto the control.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex" >
<mx:Script>
  <![CDATA[
    import ilog.maps.MapFeature;
    import ilog.maps.MapEvent;

    public function onDoubleClick(event:MapEvent):void {
      europe.fit(event.mapFeature);
    }

    public function onClick(event:MapEvent):void {
      var mf:MapFeature = event.mapFeature;
    }

    public function rollOut(event:MapEvent):void {
      europe.selectedFeatures = [];
    }

    public function rollOver(event:MapEvent):void {
      var f:MapFeature = event.mapFeature;
      europe.selectedFeatures = [f];
    }

    public function mouseWheel(event:MouseEvent):void {
      if(europe.allowNavigation)
        return;
      var mf:MapFeature = europe.selectedFeatures[0];
      if (event.delta > 0) {
        if(mf != null) {
          europe.fit(mf);
        }
      } else {
        europe.fit();
      }
    }

  ]]>
</mx:Script>

<ilog:ContinentalEuropeMap
  id="europe"
  width="100%"
  height="100%"
  toolTip="Europe"
  doubleClickEnabled = "true"
  mapItemDoubleClick = "onDoubleClick(event)"
  mapItemClick="onClick(event)"
  mapItemRollOver="rollOver(event)"
  mapItemRollOut="rollOut(event)"
  mouseWheel="mouseWheel(event)"
  allowNavigation="false"
  filterEvents="true"
  zoomableSymbols="true"  >

</ilog:ContinentalEuropeMap>

</mx:Application>
```

User interaction with maps

The maps control has some selection and navigation features built in. These are described in the following sections.

Selection

To allow selection on a given map, set the `allowSelection` property to `true`. To allow multiple selection on a given map, set the `allowMultipleSelection` property to `true`. The following code shows how to do this.

```
<ilog:WorldCountriesMap height="100%" width="100%"
   allowSelection="true" allowMultipleSelection="true" />
```

You can select as shown in the following table.

Selection actions

Function	Action
Select	Click an item in the map.
Extend a selection	Hold down the CTRL key and click the items.
Reduce a selection	Hold down the CTRL key and click an already selected item to remove it from the selection.
Clear a selection	Click anywhere on the map background, but not on map data.
Select a neighboring item.	CTRL + Arrow keys.

Navigation

A `MapBase` view can be zoomed and panned to display a specific part of the map. The navigation features can be controlled by the `allowNavigation` property of the map. The following code shows how to do this.

```
<ilog:WorldCountriesMap height="100%" width="100%"
   allowNavigation="true" />
```

The available actions are described in the following table.

Navigation actions

Action	Mouse	Keyboard
Zoom in	Rotate the mouse wheel forward.	+
Zoom out	Rotate the mouse wheel backward.	-
Pan	Click and drag.	Arrow keys

Using effects in a map

You can use IBM ILOG Elixir to make interesting and engaging maps. Important factors to consider are how user interactions with the applications trigger effects and events. Map features are `UIComponent`s and any standard effects can be used with them.

The following example shows a zoom effect when the pointer enters a country.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
  <![CDATA[
    import mx.effects.Effect;
    import mx.effects.Zoom;
    import ilog.maps.MapFeature;
    import ilog.maps.MapEvent;

    private var _effects:Array = new Array();

    private function createEffect(f:MapFeature):Effect {
      var z:Zoom = new Zoom();
      z.zoomWidthFrom = 1;
      z.zoomWidthTo = 1.5;
      z.zoomHeightFrom = 1;
      z.zoomHeightTo = 1.5;
      z.duration = 1000;
      return z;
    }

    private function play(s:MapFeature, io:Boolean):void {
      var z:Effect = _effects[s];
      if(z == null) {
        z = createEffect(s);
        _effects[s] = z;
      }
      if(z.isPlaying) {
        z.reverse();
      } else {
        z.play([s], io);
      }
    }

    public function rollOver(event:MapEvent):void {
      var m:MapFeature = event.mapFeature;
      m.parent.setChildIndex(m, m.parent.numChildren - 1);
      play(m, false);
    }

    private function rollOut(e:MapEvent):void {
      var m:MapFeature = e.mapFeature;
      play(m, true);
    }

  ]]>
</mx:Script>
<ilog:ContinentalEuropeMap
  id="europe"
  width="100%"
  height="100%"
  toolTip="Europe"
  mapItemRollOver="rollOver(event)"
  mapItemRollOut="rollOut(event)" />

</mx:Application>
```

Using built-in effects in a map

Describes the two built-in effects: color transition effect and fit effect.

In this section

Color transition effect
Describes the color transition effect.

Fit effect
Describes the fit effect.

Color transition effect

The color transition effect allows a smooth color transition when the color of a state or a country is changed. This effect only works for property values that are of type `SolidColor`.

You can activate this effect by setting the `animationDuration` style property to a positive value. This property is the duration of the effect, in milliseconds. When the `animationDuration` style property is set, the `setStyle()` method of the map or of the `mapFeature` triggers the transition effect, provided that the old and new values are of the type `SolidColor`.

Using this effect you can provide a smooth transition between the `fill`, `highlightFill`, `stroke`, and `highlightStroke` style properties.

The following code shows how to trigger a color transition effect for the entire map.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import mx.graphics.SolidColor;
      import ilog.maps.MapBase;
      import ilog.maps.MapEvent;
      private function click(e:MapEvent):void {
        var sc:SolidColor = new SolidColor(0xff0000);
        map.setStyle(MapBase.FILL, sc);
      }
    ]]>
  </mx:Script>
  <ilog:WorldCountriesMap mapItemClick="click(event)" id="map"
    animationDuration="1000" width="100%" height="100%" />
</mx:Application>
```

Fit effect

The fit effect provides a smooth zoom in/zoom out when the map is fitted to the component or when the map is fitted to a country or a state. The transition is automatically performed by a call to the `fit()` method with `(mf:MapFeature = null)`, if the `animationDuration` style property is set to a positive value.

The following code shows how to trigger a fit transition effect when the user clicks the map.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
  <![CDATA[
    import ilog.maps.MapFeature;
    import ilog.maps.MapEvent;
    private function click(e:MapEvent):void {
      var mf:MapFeature = e.mapFeature;
      map.fit(mf);
    }
  ]]>
</mx:Script>
  <ilog:WorldCountriesMap width="100%" height="100%" id="map"
    animationDuration="2000" mapItemClick="click(event)"/>
</mx:Application>
```

Using extra map features

Describes the following extra map features: drawing, longitude/latitude API.

In this section

Drawing
Describes how to add drawing with the `drawingCanvas` property.

Longitude/latitude methods
Describes how to use the fit and extent methods for longitude and latitude coordinates.

Synchronizing a component with a map
Describes how to use the `matrix` property of the `MapBase` object to synchronize a component with a map.

Drawing

You can put additional drawing onto a map using the `drawingCanvas` property of a map. Using this Canvas often requires a coordinate transformation API to convert coordinates. This can be achieved by using longitude/latitude coordinates and the related conversion methods. The drawing is made on the drawing canvas and the coordinates are converted using the longitude/latitude conversion methods `canvasToLatLong` and `latLongToCanvas`.

The following example draws a line between the mouse pointer and the barycenter of the nearest map feature.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
 creationComplete="init()">
<mx:Script>
  <![CDATA[
    import mx.controls.Button;
    import ilog.maps.MapFeature;
    import mx.graphics.Stroke;
    import mx.containers.Canvas;

    private var _label:Button;
    private var _stroke:Stroke = new Stroke(0xff0000, 1);

    internal function init():void {
       map.addEventListener(MouseEvent.MOUSE_MOVE, mouseMove);
    }

   // mouse is moving
   internal function mouseMove(e:MouseEvent):void {
     var m:Point = new Point(e.stageX, e.stageY);
     var ml:Point = map.drawingCanvas.globalToLocal(m);
     // drawing canvas to long/lat conversion
     var p:Point = map.canvasToLatLong(ml);
     var f:MapFeature = nearest(p);
     if(f != null) {
       var b:Point = f.barycenter;
       var c:Canvas = map.drawingCanvas;
       var g:Graphics = c.graphics;
       g.clear();
       _stroke.apply(g);
       // convert back to canvas coordinates
       p = map.latLongToCanvas(p);
       // convert to canvas coordinates
       b = map.latLongToCanvas(b);
       // draw the line (and a circle)
       g.moveTo(p.x, p.y);
       g.lineTo(b.x, b.y);
       g.drawCircle(b.x, b.y, 10);
       // put a label at mouse location
       if(_label == null) {
         _label = new Button();
         _label.width = 100;
         _label.height = 30;
         map.addChild(_label);
       }
       _label.label = map.getString(f.key);
       m = map.globalToLocal(m);
       _label.move(m.x, m.y);
      }
    }

    // pseudo distance between two points
    internal static function distance(p1:Point, p2:Point):Number {
      var dx:Number = p1.x - p2.x;
      var dy:Number = p1.y - p2.y;
        return dx * dx + dy * dy;
    }
    // find the nearest map feature
    internal function nearest(p:Point):MapFeature {
     var min:Number = Number.MAX_VALUE;
     var ret:MapFeature = null;
     for each (var f:MapFeature in map.mapFeatures) {
       var b:Point = f.barycenter;
       var d:Number = distance(p, b);
       if(d < min) {
         ret = f;
         min = d;
        }
      }
      return ret;
     }
  ]]>
</mx:Script>
  <ilog:ContinentalEuropeMap id="map" x="10" y="20" width="600" height="500"
 allowNavigation="true" />
</mx:Application>
```

Longitude/latitude methods

The following example shows how to use the `fitToArea` and `layerExtent` methods for longitude/latitude coordinates. In this example, three buttons are used to center the map on France, center the map on Paris, and fit the whole map onto the component respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application frameRate="50" xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute"
   creationComplete="parisSymbol()"
   xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
   <mx:Script>
     <![CDATA[
         import ilog.maps.MapFeature;
         import mx.controls.Button;
         import ilog.maps.MapSymbol;
         import mx.graphics.Stroke;
         import mx.containers.Canvas;

         // Paris is known to be at 2D20'26" E / 48D51'34" N
         private const parisX:Number = dms2dec(2, 20, 26);
         private const parisY:Number = dms2dec(48, 51, 34);

         // create a symbol at Paris(France) location
         public function parisSymbol():void {
           var sym:MapSymbol = new MapSymbol();
           sym.latitude = parisY;
           sym.longitude = parisX;
           var b:Button = new Button();
           b.width = 10;
           b.height = 10;
           sym.component = b;
           map.addChild(sym);
         }

         // center the map on Paris
         public function paris():void {
           var w:Number = 1;
           var b:Rectangle = new Rectangle(parisX - w/2, parisY - w/2, w, w);

           map.fitToArea(b);
         }

         // Fits the whole map
         public function full():void {
           map.fitToArea(null);
         }

         // fit to France
         public function france():void {
           var mf:MapFeature = map.getFeature("FRA");
           var r:Rectangle = mf.extent;
           map.fitToArea(r);
         }

         //convert DMS (degree/minute/second) to decimal degrees
        internal static function dms2dec(d:Number, m:Number, s:Number):Number
 {
           return d + (m + s/60)/60;
         }

     ]]>
   </mx:Script>
   <ilog:ContinentalEuropeMap animationDuration="2000"
                              id="map"
                              allowNavigation="true"
                              x="0"
                              y="25"
                              width="600"
                              height="400" />
     <mx:Button x="0" y="0" width="100" height="20"
buttonDown="france()" label="France"/>
     <mx:Button x="110" y="0" width="100" height="20"
buttonDown="paris()" label="Paris"/>
     <mx:Button x="220" y="0" width="100" height="20"
buttonDown="full()" label="All"/>

</mx:Application>
```

Synchronizing a component with a map

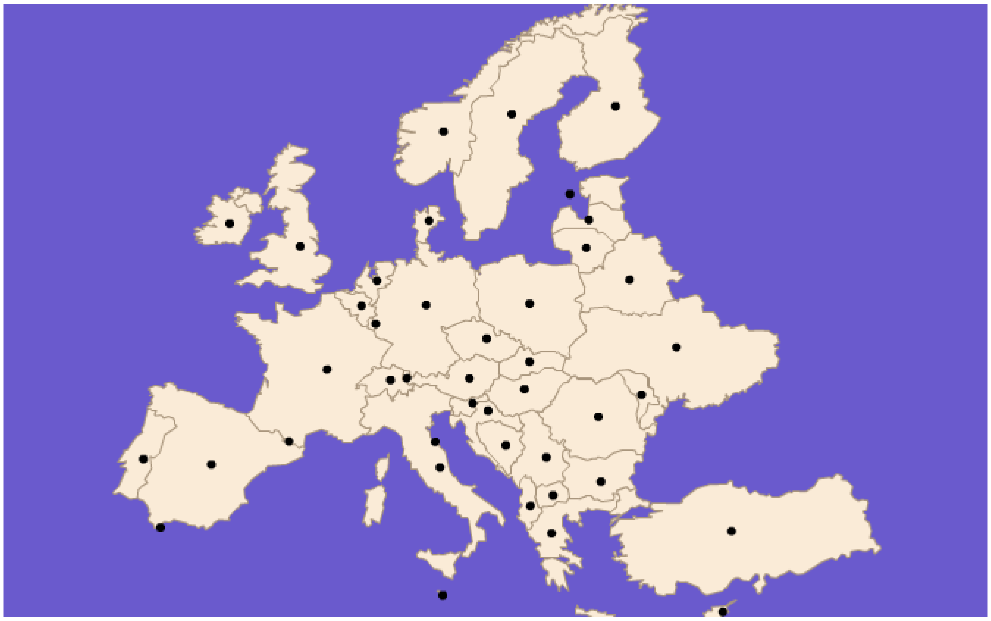
The `matrix` property specifies a `Matrix` object that reflects the current map transformation. The identity matrix applies when the map fits the component.

You can use the `matrix` property to synchronize a component with a map by listening for changes to the `matrix` property.

The following example shows how to construct a map layer as an instance of `UIComponent` on top of the map.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="absolute"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                applicationComplete="complete(event)">
<mx:Script>
  <![CDATA[
    import mx.graphics.SolidColor;
    import mx.events.PropertyChangeEvent;
    import ilog.maps.MapFeature;
    import mx.core.UIComponent;

    private var _layer:UIComponent;

    private function complete(e:Event):void {
      _layer = new UIComponent();
      map.addChild(_layer);
      map.addEventListener(PropertyChangeEvent.PROPERTY_CHANGE, synchronize);

      invalidateDisplayList();
    }

    private function synchronize(e:PropertyChangeEvent):void {
      if(e.property == "matrix") {
        _layer.transform.matrix = map.matrix;
      }
    }

    protected override function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number):void {
      super.updateDisplayList(unscaledWidth, unscaledHeight);
      if(_layer == null)
        return;
      var g:Graphics = _layer.graphics;
      g.clear();
      g.beginFill(0);
      for each (var mf:MapFeature in map.mapFeatures) {
        var p:Point = mf.barycenter;
        p = map.latLongToCanvas(p);
        p = map.drawingCanvas.localToGlobal(p);
        p = _layer.globalToLocal(p);
        g.drawCircle(p.x, p.y, 3);
      }
      g.endFill();
    }
  ]]>
</mx:Script>
  <ilog:ContinentalEuropeMap allowNavigation="true" id="map" top="10" left="10"
 bottom="20" right="10" />
</mx:Application>
```

This example creates a layer that contains a dot on each country. The positions and sizes of the dots remain synchronized with the countries on the map when you resize the map.



Generating custom maps

Describes how to use the IBM® ILOG® Elixir Custom Map Converter tool for generating custom maps from Shapefiles.

In this section

Getting started
Describes how to run the IBM® ILOG® Elixir Custom Map Converter tool and how to create a sample custom map of US states.

Getting to know the IBM® ILOG® Elixir Custom Map Converter tool
Describes the IBM® ILOG® Elixir Custom Map Converter tool and its parameter tabs, views, and functions.

Getting started

Describes how to run the IBM® ILOG® Elixir Custom Map Converter tool and how to create a sample custom map of US states.

In this section

Running the IBM® ILOG® Elixir Custom Map Converter tool
Describes how to run the IBM® ILOG® Elixir Custom Map Converter tool

Creating a custom map
Explains how to create a custom map. The example used is a map of US states.

Excluding countries or states from a custom map
Explains how to exclude features from a custom map.

Running the IBM® ILOG® Elixir Custom Map Converter tool

To run the IBM® ILOG® Elixir Custom Map Converter tool on Microsoft® Windows® , choose the following entry on the start menu: **Start>All Programs>IBM ILOG> IBM ILOG Elixir 2.5>IBM® ILOG® Elixir Custom Map Converter**.

On Mac OS® or Linux® , run the IBM® ILOG® Elixir Custom Map Converter tool from the `<installation-dir>/samples` directory.

Creating a custom map

A custom map is created from an input Shapefile.

The IBM® ILOG® Elixir Custom Map Converter must be running.

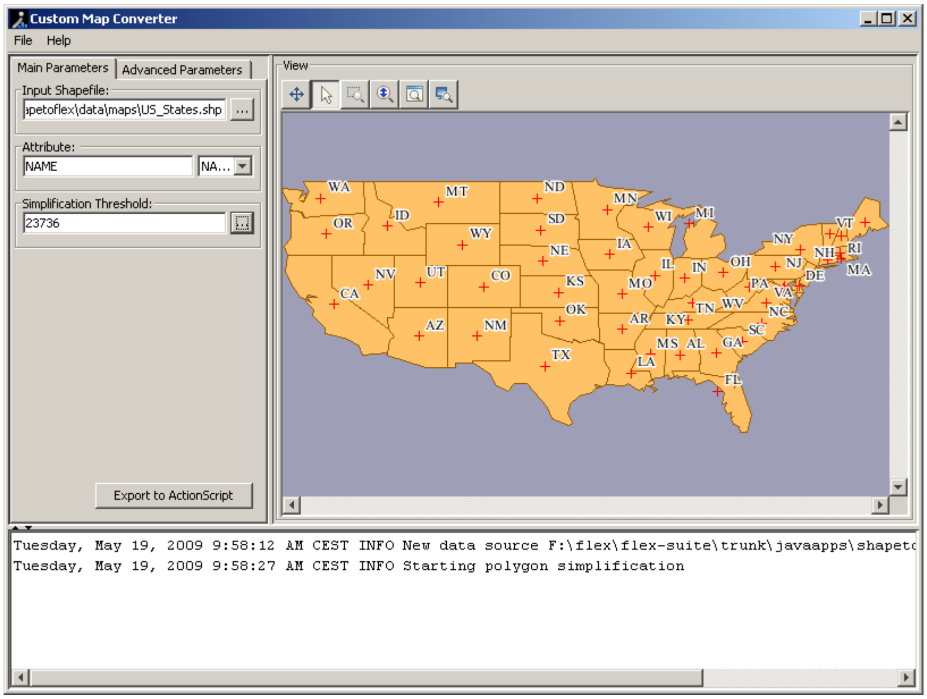
The input Shapefile must consist of the following files:

- `.shp` containing geometries (used to draw the map)
- `.dbf` containing metadata (shown as tooltips on map features)
- `.shx` containing indexing information

To create a custom map of US states:

1. In the `Main Parameters` tab, set the `Input Shapefile` parameter by browsing to select the `US_States.shp` file or drag the `US_States.shp` file into the View pane. The tool loads the map data and displays the map in the View pane. You can see the map features and the calculated barycenters of the features. In the map of US states, the features are states. The attribute list reflects the contents of the associated `.dbf` file.

2. Use the `Simplification Threshold` slider to set a mid-level threshold for simplifying the outline of the features (provinces) in the map.



Input Shapefile: apetoflex\data\maps\US_States.shp

Attribute: NAME NA...

Simplification Threshold: 23736

```
Tuesday, May 19, 2009 9:58:12 AM CEST INFO New data source F:\flex\flex-suite\trunk\javaapps\shapetc
Tuesday, May 19, 2009 9:58:27 AM CEST INFO Starting polygon simplification
```

3. Click the **Export to ActionScript** button to generate the map component in ActionScript® in the default output directory. There are two class files: `US_States.as`,

which contains the geometries of the map features, and `US_StatesMap.as` which contains the main class. The default output directory is your user directory. For example, on Microsoft® Windows® platforms, this is `C:\Documents and Settings\`*<username>* .

4. Click **File>Save Project** and specify a project file name. A project file is an XML file that associates all files and parameter settings related to the custom map. Once you have created a project file, you can reload it later to see your custom map again with the same data, parameter settings, and barycenter positions.

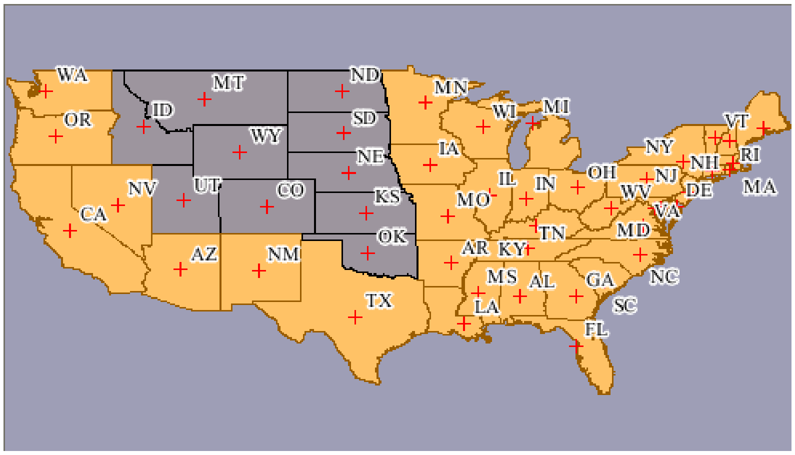
5. Once you have created your ActionScript classes, copy the files to your Flex® project and add the classes to the classes to be compiled. The new map is now ready to use.

Excluding countries or states from a custom map

To exclude some features from the generated map:

1. Select the features you want to exclude by dragging the mouse pointer around them.
2. Right-click and choose **Hide Selection** from the menu.
3. You can modify the excluded features by changing your selection, right-clicking and choosing **Show Selection** or **Hide Selection** as many times as you like.

Hidden features are shown in grey in the IBM® ILOG® Elixir Custom Map Converter.



Features that are hidden when you generate a map will be excluded from the generated map.

Getting to know the IBM® ILOG® Elixir Custom Map Converter tool

Describes the IBM® ILOG® Elixir Custom Map Converter tool and its parameter tabs, views, and functions.

In this section

Main Parameters tab
Describes the main parameters.

Advanced Parameters tab
Describes the advanced parameters.

Coordinate systems supported
Lists the coordinate systems supported by the IBM® ILOG® Elixir Custom Map Converter.

View pane
Describes the `View` pane where the map is displayed including the barycenters, the tooltips, and the possible interactions.

Project file
Describes the project file and how to create, save, and reload it.

Main Parameters tab

The red fields in parameter tabs are mandatory. Other fields are optional.

The main parameters are:

Input Shapefile
A Shapefile is a set of files containing cartographic data.

An input Shapefile for the IBM® ILOG® Elixir Custom Map Converter must have the following files:

- `.shp` containing geometries (used to draw the map)
- `.dbf` containing metadata (shown as tooltips on map features)
- `.shx` containing indexing information

There can be further indexing files but these are not used by the IBM® ILOG® Elixir Custom Map Converter. The main file is the `.shp` file. Browse to select this file.

Attribute
The list of attributes reflects the contents of the `.dbf` file. When you select an attribute, its values are displayed for each map feature on the map. For the display to be interesting, select an attribute that has different values for different map features.

Simplification threshold
To make the outlines of map features less complex, the IBM® ILOG® Elixir Custom Map Converter can apply the Douglas-Peucker algorithm for polyline simplification. There is a trade-off between the precision of the outline and the quantity of data (size of ActionScript® class): a high threshold means a greater loss of precision in the outlines but a smaller ActionScript class; a low threshold means a more precise outline for each feature but a larger ActionScript class. You can select the level you prefer.

Warning: A very high threshold can make some features of the map completely disappear.

The **Export to ActionScript** button starts the Export operation.

This operation creates two ActionScript classes with default names:

- `<Shapefile_name>.as` containing the map geometries
- `<Shapefile_name>Map.as` containing the main class

The properties, styles, and events of the main class are documented in the main class file.

Advanced Parameters tab

All advanced parameters are optional.

The advanced parameters are:

ActionScript Class Name
Several choices are offered for the geometries class name in ActionScript® based around the name of the shapefile and IBM ILOG Elixir. The associated main class will have the same name with the `Map` suffix.

Output Directory
By default, the output directory is your user directory, also called your home directory. On Microsoft® Windows® machines, this is `C:\Documents and Settings\` *<username>* ; on Mac® machines, click the **Home** icon in the **Finder** toolbar to access your home directory; on Linux® machines, every user has a user directory that is at `/home/` *<username>* .

BaryCenter Shapefile
A barycenter file is created if you modify a barycenter of the map manually. The barycenter file is created next to the main Shapefile. When you reload a project or a Shapefile, the IBM® ILOG® Elixir Custom Map Converter looks for barycenter files in the same directory as the main Shapefile. If it finds any, it loads the barycenters from these files.

Source Coordinate System
By default, this is the geographic system of latitude and longitude. You can select a different projection name from the list.

Target Coordinate System
By default, this is the geographic system of latitude and longitude. You can select a different projection name from the list. When you select a target projection, the map in the `View` pane changes to show the result.

For each coordinate system, there is cartography information:

Advanced
Select the **Advanced** option to view and set the cartography parameters associated with the source or target projection, depending on which tab is selected. Each coordinate system is presented as a projection name and parameters, plus its datum (if applicable) and ellipsoid.

Coordinate systems supported

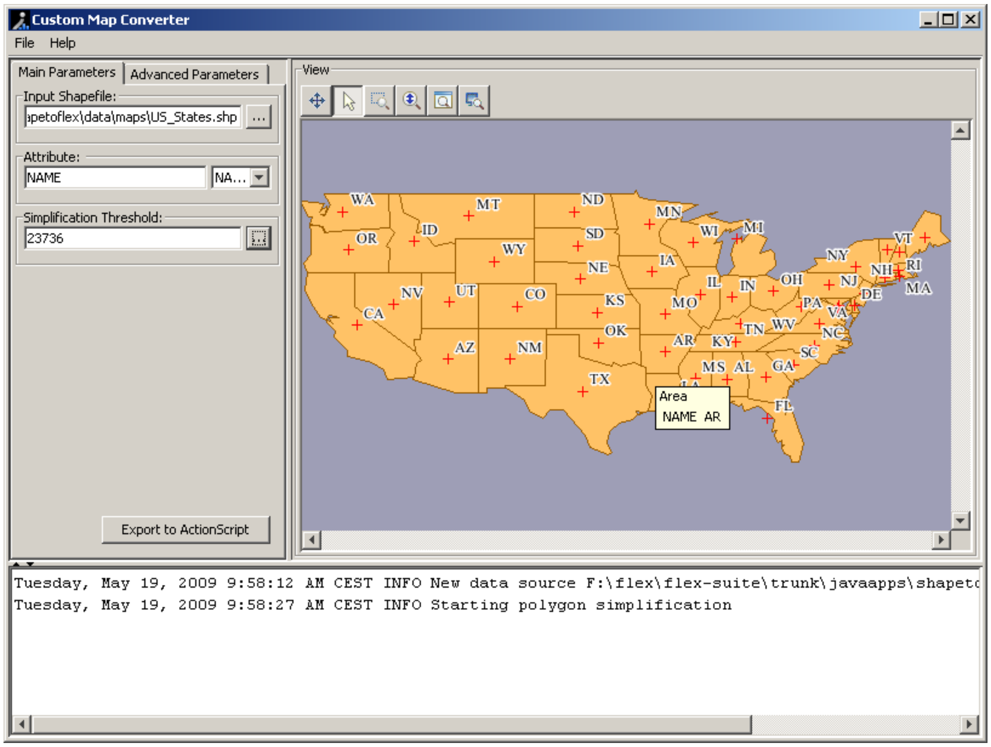
The IBM® ILOG® Elixir Custom Map Converter supports the following coordinate systems:

- Geographical
- Albers Equal Area
- Azimuthal Equidistant
- Cassini
- Cylindrical Equal Area
- Eckert IV and Eckert VI
- Equidistant Cylindrical Projection
- French Lambert
- Gnomonic
- Lambert Azimuthal Equal Area, Lambert Conformal Conic, and Lambert Equal Area Conic
- Mercator, Oblique Mercator, and Transverse Mercator
- Miller Cylindrical
- Mollweide
- Orthographic
- Polyconic
- Robinson
- Sinusoidal
- Stereographic
- Universal Polar Stereographic and Universal Transverse Mercator
- Wagner IV

View pane

The barycenters of the map features are calculated approximately. You can move a barycenter by clicking and dragging it to a new location. When you export to ActionScript® or save as a project, the new barycenter positions are saved in a `.shp` file and will be reloaded whenever you reload the map from the corresponding Shapefile or a project file.

Hover over a barycenter to see the tooltip associated with a barycenter or over a map feature to see the tooltip associated with a map feature. The information shown in the tooltip comes from the metadata in the `.dbf` file. An example of a barycenter tooltip is shown in the following figure.



Input Shapefile: apetoflex\data\maps\US_States.shp

Attribute: NAME

Simplification Threshold: 23736

```
Tuesday, May 19, 2009 9:58:12 AM CEST INFO New data source F:\flex\flex-suite\trunk\javaapps\shapetc
Tuesday, May 19, 2009 9:58:27 AM CEST INFO Starting polygon simplification
```

The available interactions with the map in the `View` pane are listed in the following table.

Icon or key	Interaction
✥	Pan the map in any direction.
↖	Select map feature.
🔍	Zoom in on map.
🔍	Zoom in continuously on the map.
🔍	Fit the map to the View pane.
🔍	Magnify the area around the cursor by 4x.
CTRL+Arrow key	Select neighboring map feature in any direction.

Project file

The project file contains references to all the items and parameter settings related to the custom map.

To save a project file, click **File>Save Project** and browse to the location. You can save to a new or existing file.

To reload a project from a project file, click **File>Load Project** and browse to select the project file.

To clear the `View` pane and all parameter settings, click **File>New Project**.

OLAP and pivot charts

Describes the use of IBM ILOG Elixir OLAP and pivot charts with Adobe® Flex® 3.

In this section

Introduction to OLAP and pivot charts
Describes the uses of OLAP charts and pivot charts with examples.

OLAP/pivot charts architecture
Describes the OLAP and pivot chart classes with a class diagram.

OLAP charts
Describes how to create, configure, and style an OLAP chart.

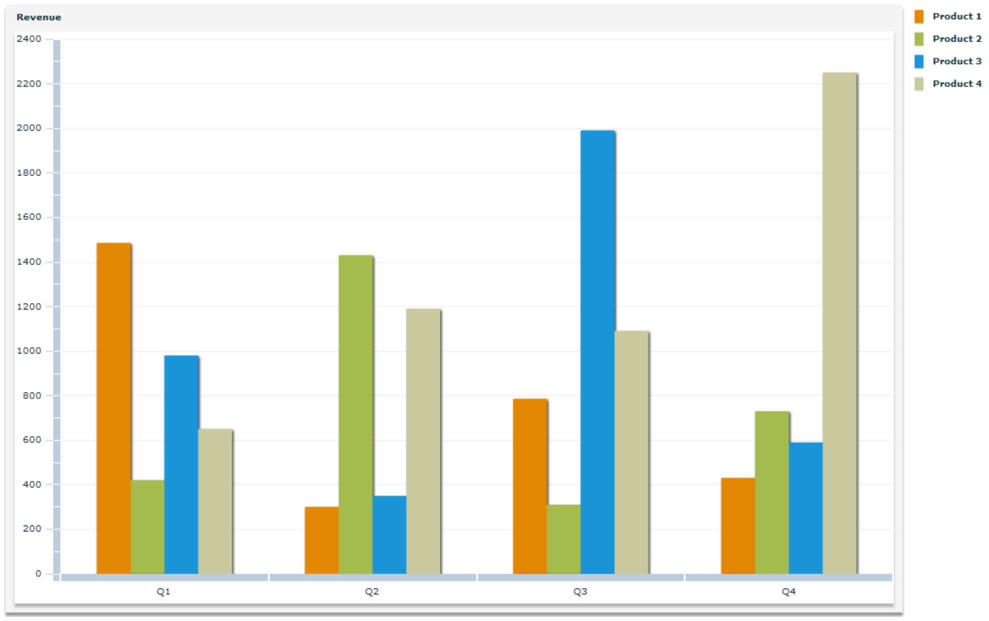
Pivot charts
Describes how to create, configure, and style a pivot chart.

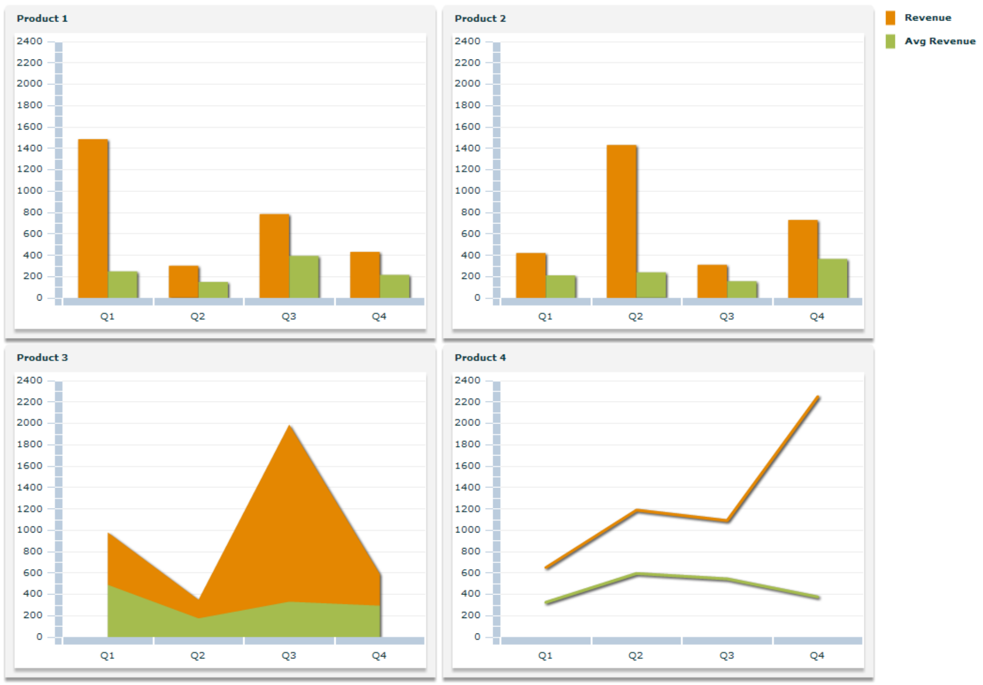
Introduction to OLAP and pivot charts

IBM ILOG Elixir lets you display the result of Online Analytical Processing (OLAP) queries in a chart component. This is a similar feature to the Adobe® Flex® `OLAPDataGrid` letting you display the result of an OLAP query in a data grid.

The IBM ILOG Elixir `OLAPChart` component can be configured to display most kinds of data series available with the Flex charting components. Depending on how it is configured, it can arrange the rows that are the result of the query as series or as chart instances in grid cells or a mixture of both.

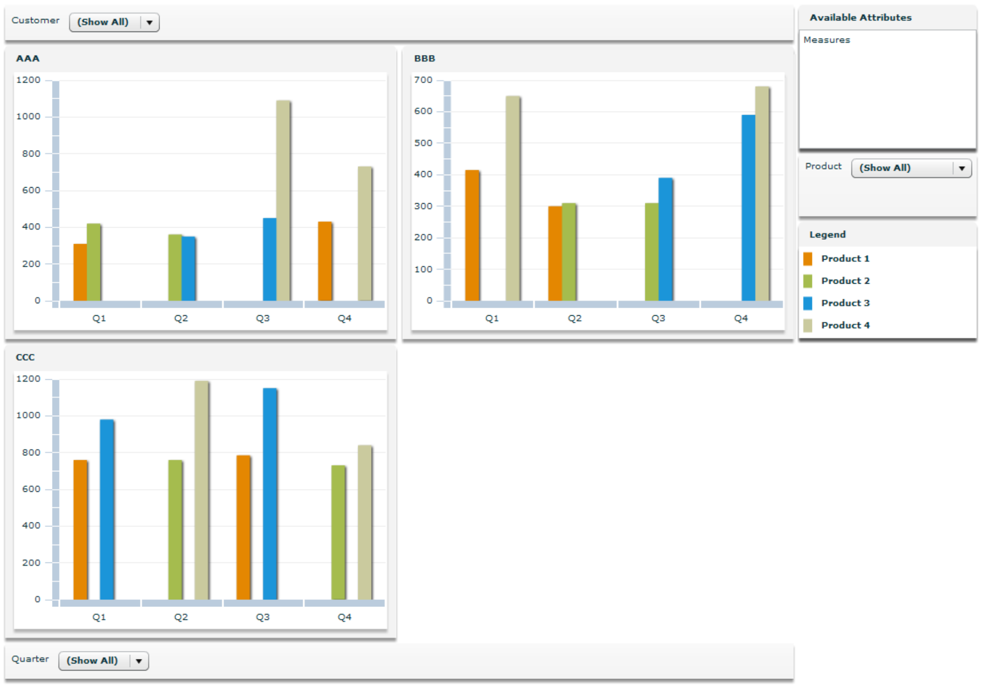
The following figures show examples of OLAP charts for a set of products.





In addition to the `OLAPChart` object, which must be configured with an OLAP query, IBM ILOG Elixir provides a pivot chart that can display OLAP data. The pivot chart does not require an OLAP query to be written; instead the end-user can configure the content of the chart using mouse gestures such as dragging the fields of interest onto the chart.

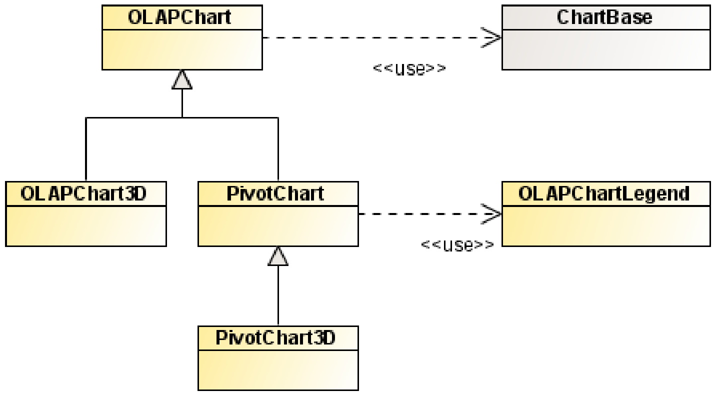
The following figure shows an example of a pivot chart.



For more information on OLAP and how to create OLAP schemas and queries that can fit an IBM ILOG Elixir component, read *Creating an OLAP Schema* and *Creating OLAP Queries* in *Advanced Data Grid Controls and Automation Tools>Creating OLAP Data Grids* in the *Flex Data Visualization Developer's Guide*.

OLAP/pivot charts architecture

The OLAP and Pivot chart classes rely on the Adobe® Flex® Builder™ Professional charting classes inheriting from `ChartBase` as shown in the following figure. This allows you to use any chart from Adobe Flex Builder Professional or IBM ILOG Elixir 3D Chart to configure your OLAP or Pivot chart.



OLAP charts

Describes how to create, configure, and style an OLAP chart.

In this section

Creating an OLAP chart
Explains how to create an OLAP chart.

Configuring an OLAP chart
Explains how to configure an OLAP chart.

Styling an OLAP chart
Explains how to style an OLAP chart.

Creating an OLAP chart

To create an `OLAPChart` component, you need to create an OLAP schema or cube that can be used to query your flat data using the Adobe® Flex® `OLAPCube` object. For details, read about *Creating an OLAP Schema* in *Advanced Data Grid Controls and Automation Tools>Creating OLAP Data Grids* in the *Flex Data Visualization Developer's Guide*.

The following code shows some simple flat data on products.

```
[ {
   product : "Product 1",
   quarter : "Q1",
   month : "January",
   year : "2008",
   country : "France",
   sales : 14
  },
   {
   product : "Product 1",
   quarter : "Q1",
   month : "January",
   year : "2008",
   country : "US",
   sales : 11
  },
...
  {
   product : "Product 2",
   quarter : "Q4",
   month : "December",
   year : "2008",
    country : "US",
    sales : 10
  },
]
```

The following code is an example of an OLAP cube that will allow you to query the summary data on products, namely the total sales by product, time, and country. This code defines product, time, and country as dimensions and sets the `aggregator` property of the `OLAPMeasure` object to make use of the `SUM` aggregator.

```
<mx:OLAPCube name="Sales" dataProvider="{flatData}" id="cube">
    <mx:OLAPDimension name="ProductDim">
    <mx:OLAPAttribute name="Product" dataField="product"/>
    <mx:OLAPHierarchy name="ProductHier" hasAll="true">
      <mx:OLAPLevel attributeName="Product"/>
    </mx:OLAPHierarchy>
  </mx:OLAPDimension>

  <mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPAttribute name="Month" dataField="month"/>
    <mx:OLAPHierarchy name="TimeHier" hasAll="true">
      <mx:OLAPLevel attributeName="Year"/>
      <mx:OLAPLevel attributeName="Quarter"/>
      <mx:OLAPLevel attributeName="Month"/>
    </mx:OLAPHierarchy>
  </mx:OLAPDimension>

  <mx:OLAPDimension name="CountryDim">
    <mx:OLAPAttribute name="Country" dataField="country"/>
    <mx:OLAPHierarchy name="CountryHier" hasAll="true">
      <mx:OLAPLevel attributeName="Country"/>
    </mx:OLAPHierarchy>
  </mx:OLAPDimension>

  <mx:OLAPMeasure name="Sales" dataField="sales"
        aggregator="SUM"/>

</mx:OLAPCube>
```

You can change the `aggregator` property of the measure to something other than `SUM`, for example, to the `AVG` aggregator to compute the average sales value instead.

Once your cube has been defined you need to write a query to extract data from the cube. The following code shows the query for the example cube.

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
  var query:OLAPQuery = new OLAPQuery();

  // on row axis, cluster by product and country dimensions
  var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
  var productSet:OLAPSet = new OLAPSet();
  productSet.addElements(cube.findDimension("ProductDim").
    findAttribute("Product").children);
  var countrySet:OLAPSet = new OLAPSet();
  countrySet.addElements(cube.findDimension("CountryDim").
    findAttribute("Country").children);
  rowQueryAxis.addSet(productSet.crossJoin(countrySet));

  // on column axis, cluster, by Year, Quarter
  var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
  var yearSet:OLAPSet = new OLAPSet();
  yearSet.addElements(cube.findDimension("TimeDim").
    findAttribute("Year").children);
  var quarterSet:OLAPSet= new OLAPSet();
  quarterSet.addElements(cube.findDimension("TimeDim").
   findAttribute("Quarter").children);
  colQueryAxis.addSet(yearSet.crossJoin(quarterSet));

  return query;
}
```

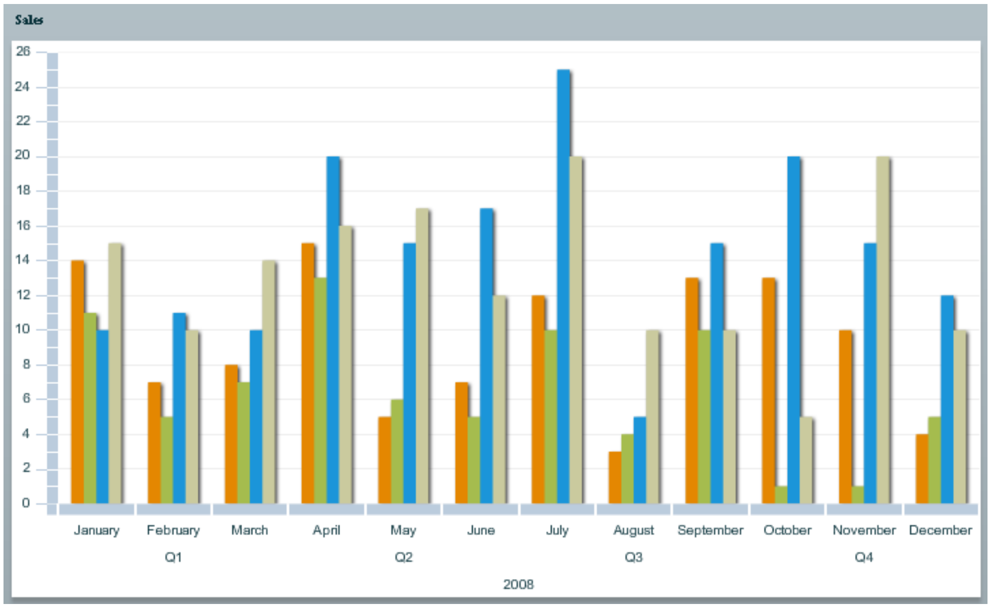
You need to make sure the query is run after the cube has been completed and that a `showResult` function is called. The following code shows how to specify a function to execute the query and how to set up a result handler.

```
<mx:Script>
  <![CDATA[
  private function runQuery(event:Event):void {
    var query:IOLAPQuery = getQuery(event.target as OLAPCube);
    // Execute the query.
    var token:AsyncToken = cube.execute(query);
    // Set up handlers for the query results.
    token.addResponder(new AsyncResponder(showResult, showFault));
  }
  ]]>
</mx:Script>
<mx:OLAPCube name="salesCube" dataProvider="{flatData}" id="cube"
  complete="runQuery(event);"/>
```

The following code shows how the `showResult` function sets the the `OLAPChart` data provider to the `OLAPResult` and how to create an `OLAPChart` object.

```
<mx:Script>
  <![CDATA[
  private function showResult(result:Object, token:Object):void {
    if (!result) {
      Alert.show("No results from query.");
      return;
    }
    olapchart.dataProvider = result as IOLAPResult;
  }
  ]]>
</mx:Script>
<ilog:OLAPChart width=”100%” height=”100%” id=”olapachart”/>
```

By default the `OLAPChart` object displays the entire result in a single `ColumnChart` using column series.



The first series (orange) corresponds to the “Product 1” sales in France, the second one (green) corresponds to the “Product 1” sales in the US, the third one (blue) corresponds to the “Product 2” sales in France and the last one to “Product 2” sales in the US. This is because the row axis of the OLAP query is configured to be a cross-join of the products and the countries.

If you want a different display, you can configure what is rendered. See *Configuring an OLAP chart*.

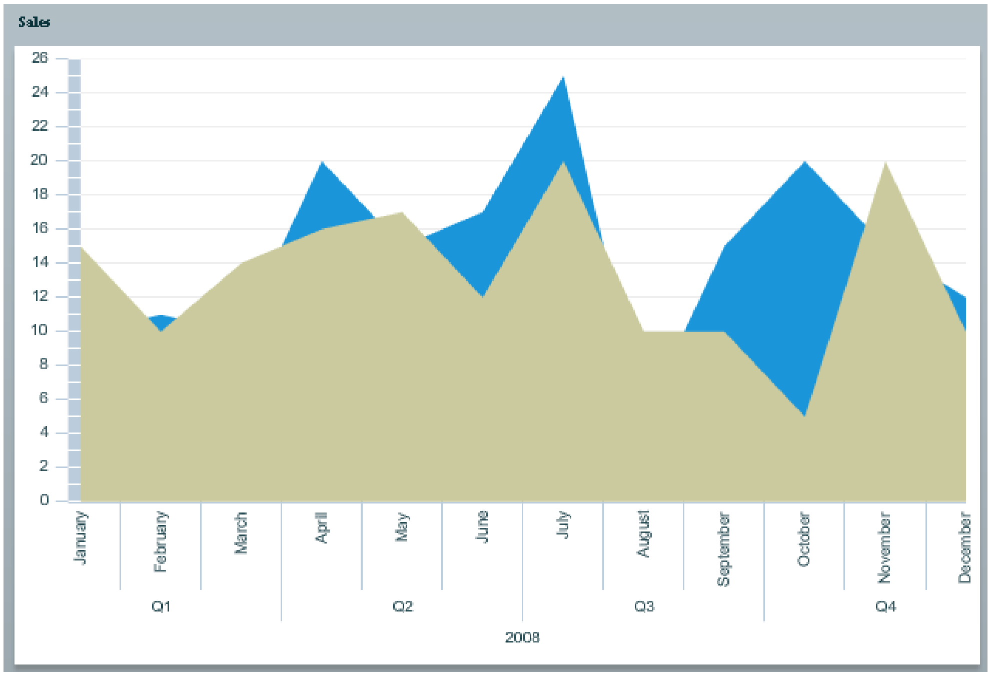
Configuring an OLAP chart

Changing chart and series types

If you want to display the result in something other than a `ColumnChart` you have to configure the `OLAPChart`. The following example shows how to use an `AreaChart` instead of a `ColumnChart`.

```
<ilog:OLAPChart width=”100%” height=”100%” id=”olapachart”/>
  <ilog:categoryAxis>
    <ilog:OLAPAxis padding=”0”/>
  </ilog:categoryAxis>
  <ilog:chart>
    <mx:Component>
      <mx:AreaChart width="100%" height="100%"/>
    </mx:Component>
  </ilog:chart>
  <ilog:seriesProviders>
    <ilog:OLAPSeriesProvider>
      <ilog:series>
        <mx:Component><ilog:OLAPAreaSeries/></mx:Component>
      </ilog:series>
    </ilog:OLAPSeriesProvider>
  </ilog:seriesProviders>
</ilog:OLAPChart>
```

You can see that both the chart type (`ilog:chart`) and the series `(ilog:seriesProvider`) have to be configured to the new type of chart and corresponding series. The `OLAPAxis` padding has also been changed to align correctly with the `AreaChart`; it is no longer the default padding value which fitted nicely with `ColumnChart`.

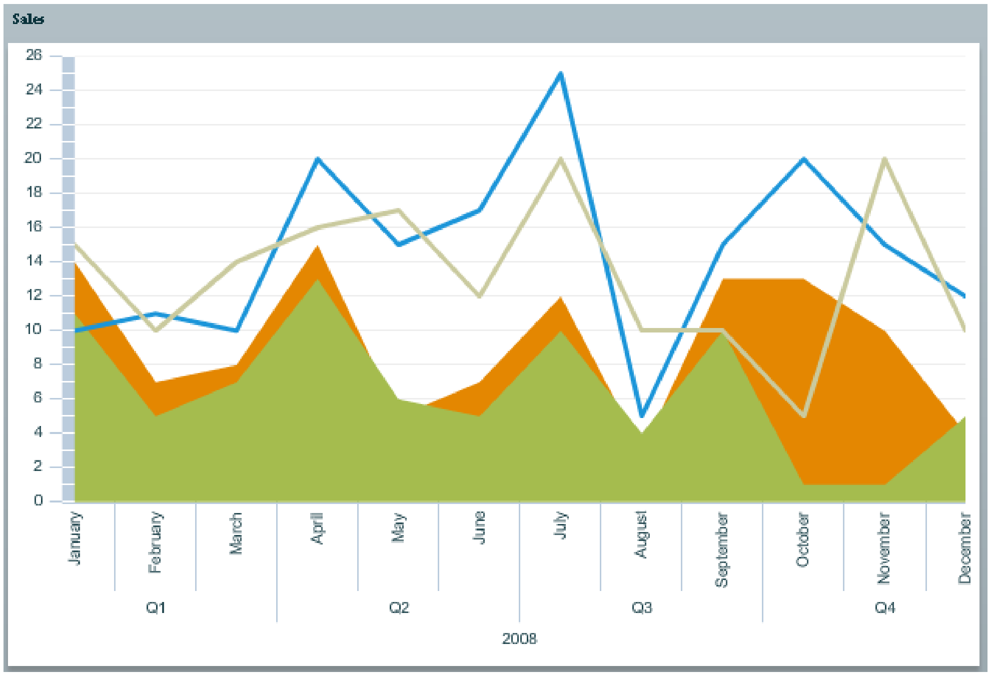


In this case there is only a single series provider without any specific selector. This means that all the series created will be instances of the given `OLAPAreaSeries` using this fallback provider. In this example, as all the series are of type area, some are hidden by others. One way to solve that problem is to make sure that “Product 1” series (France and US) are represented by `OLAPAreaSeries` but “Product 2” series (France and US) are represented by `OLAPLineSeries`.

The following example of a series providers configuration shows how series providers can be configured to make sure different series are built depending on the OLAP result row to which they correspond.

```
<ilog:seriesProviders>
  <ilog:OLAPSeriesProvider>
    <ilog:selectors>
      <ilog:OLAPElementSelector
        uniqueName="[ProductDim].[Product].[Product 1]"
        type="{OLAPElementSelector.OLAP_MEMBER}"/>
    </ilog:selectors>
    <ilog:series>
      <mx:Component><ilog:OLAPAreaSeries/></mx:Component>
    </ilog:series>
  </ilog:OLAPSeriesProvider>
  <ilog:OLAPSeriesProvider>
    <ilog:selectors>
      <ilog:OLAPElementSelector uniqueName="[ProductDim].[Product].[Product
2]"
        type="{OLAPElementSelector.OLAP_MEMBER}"/>
      <ilog:OLAPElementSelector
        uniqueName="[CountryDim].[Country].[France]"
        type="{OLAPElementSelector.OLAP_MEMBER}"/>
    </ilog:selectors>
    <ilog:series>
      <mx:Component>
        <ilog:OLAPLineSeries>
          <ilog:lineStroke>
            <mx:Stroke color="0x1B95D9" weight="3"/>
          </ilog:lineStroke>
        </ilog:OLAPLineSeries>
      </mx:Component>
    </ilog:series>
  </ilog:OLAPSeriesProvider>
  <ilog:OLAPSeriesProvider>
    <ilog:selectors>
      <ilog:OLAPElementSelector uniqueName="[ProductDim].[Product].[Product
2]"
        type="{OLAPElementSelector.OLAP_MEMBER}"/>
      <ilog:OLAPElementSelector uniqueName="[CountryDim].[Country].[US]"
        type="{OLAPElementSelector.OLAP_MEMBER}"/>
    </ilog:selectors>
    <ilog:series>
      <mx:Component>
        <ilog:OLAPLineSeries>
          <ilog:lineStroke>
            <mx:Stroke color="0xCACA9E" weight="3"/>
          </ilog:lineStroke>
        </ilog:OLAPLineSeries>
      </mx:Component>
    </ilog:series>
  </ilog:OLAPSeriesProvider>
</ilog:seriesProviders>
```

The following figure shows the chart displayed by this configuration.



Note that more specific selectors are used for “Product 2” in order to specify the stroke for the US and France series. For more details on how to configure a selector, you can read the documentation on `OLAPElementSelector`.

If there is no fallback provider (a provider without selectors) and no provider matches a given OLAP result row, then the row will not be displayed as a series in the resulting chart. For example, if there are rows for “Product 3”, they will not be displayed.

Changing clustering

In the previous example, all the rows of the OLAP query result were series in a single chart. You may want to see several charts to visualize the result better. To do this, use the clustering mechanism of the OLAP chart.

In the row axis of the OLAP query, you have several sets of data: the first set is product data and the second set is country data. By default you will get as many series as there are product/country tuples. However, you might want to see the products spread over different charts and, on each chart, have the countries represented by series. To achieve this, set the `clusteringDepth` property of `OLAPChart` to 1. As the product data is first, it will be used to create as many charts as there are products, and as the country data is second, the depth 1 will be used to build series for each country.

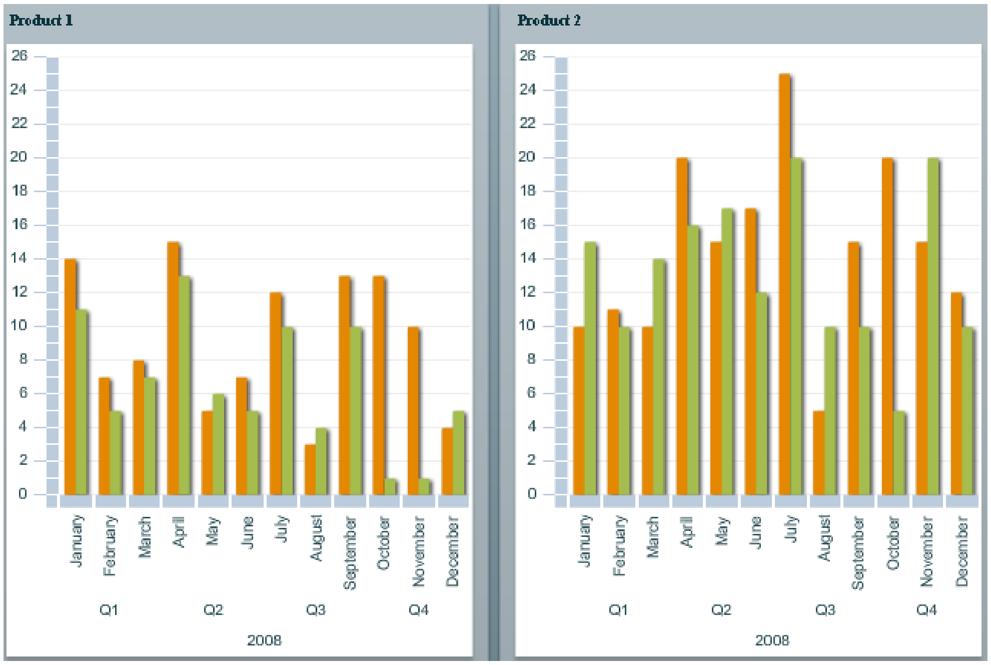
The following code shows how to set the clustering depth.

```
<ilog:OLAPChart width=”100%” height=”100%”
  id=”olapachart” clusteringDepth=”1”>
  <ilog:dataAxis>
```

```
    <mx:LinearAxis/>
  </ilog:dataAxis>
</ilog:OLAPChart>
```

You can see that in addition to the `clusteringDepth` property, the `dataAxis` property has been set on the OLAP chart. This is not mandatory, however it allows all the charts created in the OLAP chart to share the same data axis, which is vertical in this case.

The following figure shows the resulting charts. The orange series are figures for France and the green series are figures for the US, spread over two charts — the first chart for “Product 1” and the second for “Product 2”.



If you want one chart per country and one series per product instead, you just have to tweak the row axis of the OLAP query to change the order of the cross-join to have countries coming first.

```
rowQueryAxis.addSet(countrySet.crossJoin(productSet));
```

If you have a query row axis with more than two sets of data, you can set `clusteringDepth` to values greater than 1 to determine how many of the data sets are used to create charts; the remaining data sets will be used to create series.

Configuring clustering

By default the maximum number of charts displayed on a screen after clustering is 20. You can change this using the `maxNumCharts` property.

```
<ilog:OLAPChart width="100%" height="100%"
   id="olapachart" clusteringDepth="1" maxNumCharts="8"/>
```

As the number of charts displayed on a single page is limited by `maxNumCharts`, you can use the `nextPage` and `previousPage` methods of `OLAPChart` to navigate the pages as in the following XML sample.

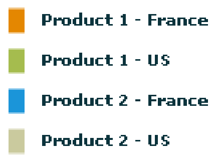
```
<mx:Button click="olapchart.previousPage()"
   enabled="{olapchart.hasPreviousPage}" label="Prev Page"/>
<mx:Button click="olapchart.nextPage()"
   enabled="{olapchart.hasNextPage}" label="Next Page"/>
<ilog:OLAPChart width="100%" height="100%"
   id="olapachart" clusteringDepth="1" maxNumCharts="8"/>
```

Legend

A specific legend object is available for `OLAPChart`. The following code shows how to display a legend.

```
<ilog:OLAPChartLegend dataProvider="{olapchart}" backgroundColor="white"/>
```

The following figure shows the resulting legend displayed.



Styling an OLAP chart

Once an OLAPChart has been configured, you might want to change its style. There are properties that let you modify the appearance of the chart itself and of the series.

OLAP chart styling

When you use clustering, the OLAP chart lays out the different charts using a Tile object. You can change the style properties of the Tile using the tileStyleName style property on the OLAPChart.

The following code shows how to set larger gaps between the charts than the default ones.

```
<mx:Style>
  .tileStyle {
    horizontalGap : 10,
    verticalGap : 10
  }
</mx:Style>
<ilog:OLAPChart width="100%" height="100%"
  id="olapachart" clusteringDepth="1"
  tileStyleName="titleStyle"/>
```

You can also replace the Container in charge of holding each chart. By default a Panel is used and the display name of the OLAP elements rendered in the chart is set to the Panel title. You can replace the panel by something else. If you want the title to be available, the replacement container must implement the IDataRenderer interface. In this case, the title will be available in the data property of the holder.

If you do not want any intermediate holder between the tile and the chart you can just set the holder property to null.

```
<ilog:OLAPChart width="100%" height="100%"
  id="olapachart" clusteringDepth="1">
  <ilog:holder>{null}</ilog:holder>
</ilog:OLAPChart>
```

If you want the OLAP chart to display 3D charts instead of 2D charts, use the 3D version of the chart.

```
<ilog:OLAPChart3D width="100%" height="100%" id="olapachart"/>
```

When you use the 3D version of the OLAP chart, you must use 3D OLAP series in series providers instead of the 2D versions.

OLAP axis styling

By default, the OLAPChart uses instances of OLAPAxisRenderer to render the category axis of the charts. This specific renderer can display a hierarchy of data sets provided by the OLAP result columns on a single axis.

The `OLAPAxisRenderer` can rotate labels to display them fully. If you want to enable this possibility, you will have to provide an embedded font to the axis renderer as shown in the following code.

```
<mx:Style>
  @font-face {
    src: local("sansserif");
    fontFamily: "myFont";
  }
</mx:Style>
<ilog:OLAPChart width=”100%” height=”100%”
  id=”olapachart” font-family=”myFont”/>
```

To go further, the following example shows how to:

♦ Flip the vertical labels on the axis (bottom-to-top instead of top-to-bottom)

♦ Have a different tick stroke instead of the default one

♦ Display a title on the axis

```
<ilog:OLAPChart id="olapchart" width="100%" height="100%"
  fontFamily="MyFont">
  <ilog:categoryAxis>
    <ilog:OLAPAxis title="Axis Title"/>
  </ilog:categoryAxis>
  <ilog:chart>
    <mx:Component>
      <mx:ColumnChart width="100%" height="100%" showDataTips="true">
         <mx:horizontalAxisRenderer>
           <ilog:OLAPAxisRenderer flipVertical="true">
             <ilog:tickStroke>
               <mx:Stroke color="gray" weight="3"/>
             </ilog:tickStroke>
           </ilog:OLAPAxisRenderer>
         </mx:horizontalAxisRenderer>
      </mx:ColumnChart>
    </mx:Component>
  </ilog:chart>
</ilog:OLAPChart>
```

See `OLAPAxis` and `OLAPAxisRenderer` in the reference manual for more details.

OLAP series styling

As with regular charts, OLAP chart series can be styled to adapt the rendering to the data.

The following example shows how to color the monthly sum of sales red if the value is less than four.

```
<mx:Script>
  <![CDATA[
    import mx.graphics.SolidColor;
    import mx.olap.IOLAPAxisPosition;
    import mx.olap.IOLAPCell;
    import mx.graphics.IFill;

    public function fillFunction(cell:IOLAPCell,
                                 column:IOLAPAxisPosition,
                                 row:IOLAPAxisPosition):IFill {
      if (cell.value < 4)
        return new SolidColor(0xff00000);
      else
        // when null is returned it falls back to the default fill
        return null;
   }
  ]]>
</mx:Script>
<ilog:OLAPChart id="olapchart" width="100%" height="100%">
  <ilog:seriesProviders>
    <ilog:OLAPSeriesProvider>
      <ilog:series>
        <mx:Component>
          <ilog:OLAPColumnSeries
                olapFillFunction="{outerDocument.fillFunction}"/>
        </mx:Component>
      </ilog:series>
    </ilog:OLAPSeriesProvider>
  </ilog:seriesProviders>
```

Pivot charts

Describes how to create, configure, and style a pivot chart.

In this section

Creating a pivot chart
Explains how to create a pivot chart.

Configuring a pivot chart
Explains how to configure a pivot chart.

Styling a pivot chart
Explains how to style a pivot chart.

Managing PivotChart events
Explains how to make use of the events generated by a pivot chart in your listeners.

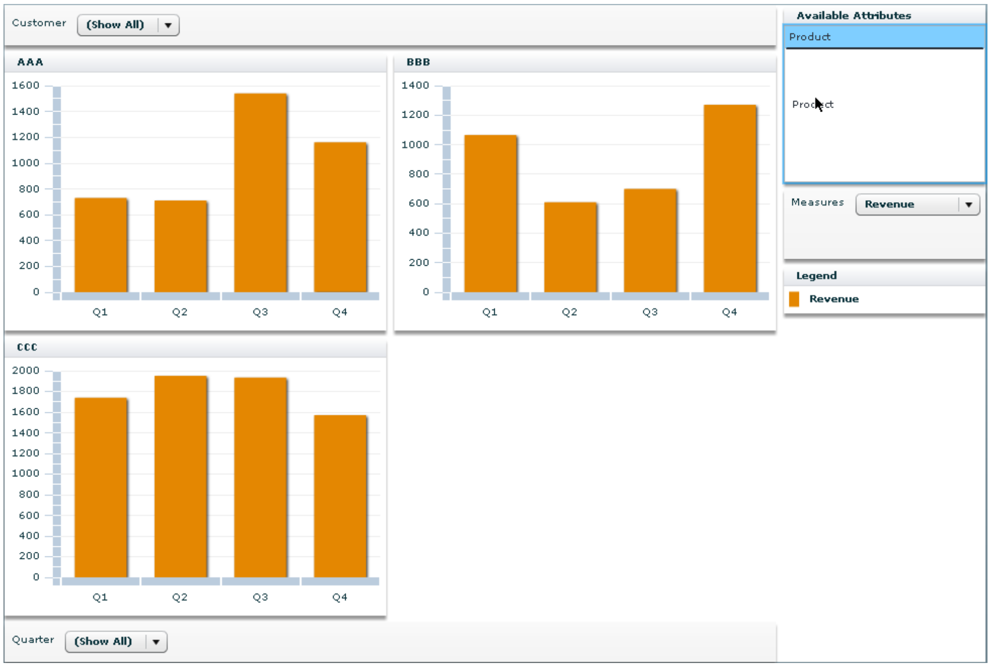
Creating a pivot chart

The major difference between an `OLAPChart` and a `PivotChart` is that the OLAP chart is configured from the result of an OLAP query whereas the pivot chart is configured from the OLAP cube itself. The end user will build the query using mouse gestures in the final application from the information introspected in the OLAP cube. This means that as for the OLAP chart, the first thing to do to create a pivot chart is to create a cube that fits your data. See *Creating an OLAP chart*.

Once you have an OLAP cube, you bind it on the PivotChart cube property as shown in the following code.

```
<mx:OLAPCube name="Sales" dataProvider="{flatData}" id=”olapcube”>
  [ ... ]
</mx:OLAPCube>
<ilog:PivotChart width=”100%” height=”100%” cube=”{olapcube}” />
```

You do not have to create the query yourself, as the attributes in the OLAPCube will be accessible in the application for the end user to configure the display. See the following example.



The end user can drag the attributes from the list of attributes to one of the areas used to configure the pivot chart:

- At the bottom, the category area. All attributes dropped in that area will be used to configure the category axis of the `PivotChart`. If several attributes are there, a cross-join will be made between them in the order in which they appear.

- At the top, the chart area. All attributes dropped in that area will be used to configure how many charts will be created by the `PivotChart`. If several attributes are there, a cross-join will be made between them in the order in which they appear.
- On the right, the series area. All attributes dropped in that area will be used to configure how many series will be created per chart by the `PivotChart`. If several attributes are dropped, a cross-join will be made between them in the order in which they appear.

When an attribute is dragged to an area, it displays a dropdown list of the available choices for the given attribute. You can choose either to show all the possibilities for an attribute (for example “Product 1” and “Product 2” for the `Product` attribute) using the `(Show All)` option or only some of them. When an attribute is dragged to the chart or series area, another option called `(All)` is available. In this case all the values of the given attribute are aggregated and displayed as a single value.

When you drag the `Measures` attribute, you can choose which measures you want to be displayed in the pivot chart. These values come from the “Measures” dimension of the OLAP cube.

Another feature available for the pivot chart but not for the OLAP charts is navigation between pages when the number of charts exceeds the specified limit for one page. Navigation is implemented as arrow buttons which appear automatically above the chart area when needed.

Configuring a pivot chart

Changing chart and series types

The `PivotChart` class inherits from the `OLAPChart` class. This means that you can use the exact same methodology to change chart and series types, see *Changing chart and series types*.

Other configurations described below are more specific to the `PivotChart` object.

Pivot chart query configuration

The `PivotChart` query is meant to be configured by the end user using mouse gestures. However in some cases you might want to present the `PivotChart` with a default query at launch time. This is possible using the APIs of the `PivotChart`.

Suppose you want to display the data by default in exactly the same manner as in the OLAP “Changing Clustering” section: a chart per product and a series per country, with the year, quarter, and month displayed on the category axis. The following exmple shows how to set the `charts`, `series`, and `categoryAttributes` properties to the right set of `PivotAttribute` objects.

```
<mx:Script>
  <![CDATA[
    import mx.collections.IList;
    import ilog.charts.PivotAttribute;
    public function getAttributes(elements:IList, attributes:Array):Array
    {
      var result:Array = []
      for each (var attribute:PivotAttribute in attributes) {
        if (elements.getItemIndex(attribute.element) != -1)
          result.push(attribute);
      }
      return result;
    }
  ]]>
</mx:Script>
<ilog:PivotChart id="pivotchart" width="100%" height="100%" cube="{cube}"
  chartsAttributes="{getAttributes(cube.findDimension('CountryDim').attributes,

                                   pivotchart.attributes)}"
  seriesAttributes="{getAttributes(cube.findDimension('ProductDim').attributes,

                                   pivotchart.attributes)}"
  categoryAttributes="{getAttributes(cube.findDimension('TimeDim').attributes,

                                   pivotchart.attributes)}">
  <ilog:dataAxis>
    <mx:LinearAxis/>
  </ilog:dataAxis>
</ilog:PivotChart>
```

Other properties

Several properties are available in addition to the `PivotChart` query configuration. You can get a full list by looking at the `PivotChart` class.

The most useful other properties are:

enable

You can disable interaction on the `PivotChart` by setting the `enable` attribute to `false`.

```
<ilog:PivotChart id="pivotchart" width="100%" height="100%"
  cube="{cube}" enable="false"/>
```

showLegend

`PivotChart` has an embedded `OLAPChartLegend` visible by default on the right side of the component. If you want to switch it off you can set the `showLegend` style property to `false`.

```
<ilog:PivotChart width="100%" height="100%" cube="{olapcube}"
  showLegend="false"/>
```

queryFactory and setFactory

`PivotChart` uses the Adobe® Flex® Builder™ Professional OLAP in-memory implementation by default. You may want to replace it by your own implementation. To do so, you can use the `queryFactory` and `setFactory` properties to set your own implementation classes as shown in the following example.

```
<ilog:PivotChart width=”100%” height=”100%”
  cube=”{olapcube}”
  setFactory="{new ClassFactory(mx.olap.OLAPSet)}"
  queryFactory="{new ClassFactory(mx.olap.OLAPQuery)}"
```

Styling a pivot chart

The `PivotChart` class inherits from the `OLAPChart` class. This means that you can use all the style properties available on the OLAP chart on the pivot chart. For details, see *Styling an OLAP chart* . In particular, just as for the OLAP chart you can use a 3D version of the pivot chart by using the `PivotChart3D` instead of the PivotChart one.

```
<ilog:PivotChart3D width="100%" height="100%" cube="{olapcube}"/>
```

When you use the 3D version of the pivot chart you must use 3D OLAP series in series providers instead of the 2D versions.In addition, `PivotChart` provides additional styling possibilities to make sure the subcomponents of the pivot chart (drop areas, legend, and so on) can be styled.

Subcomponent layout

You can change the gap between the subcomponents of the pivot chart by using the `verticalGap` and `horizontalGap` style properties.

```
<ilog:PivotChart id="pivotchart" width="100%" height="100%" cube="{cube}"
  horizontalGap="50" verticalGap="50"/>
```

Also, as for the `OLAPChart` component, you can use the `tileStyleName` property to configure the layout of the tile.

```
<mx:Style>
  .tileStyle {
    horizontalGap : 10,
    verticalGap : 10
  }
</mx:Style>
<ilog:PivotChart id=”pivotchart” width=”100%” height=”100%” cube=”{cube}”
  horizontalGap=”10” verticalGap=”10” tileStyleName=”titleStyle”/>
```

Subcomponent styling

Each of the subcomponents created by the pivot chart has either a style name on the `PivotChart` class or styling properties on that class for styling purposes. The following example shows how to change the styling of the legend and the drop areas.

```
<mx:Style>
  .myLegendStyle {
    backgroundColor : #ffffff;
    labelPlacement: top;
  }
</mx:Style>
<ilog:PivotChart id="pivotchart" width="100%" height="100%" cube="{cube}"
  legendStyleName="myLegendStyle"
  dropAreaCornerRadius="4"
  dropAreaBorderColors="[ 0xffffff,  0xffffff ]" />
```

Managing PivotChart events

The `PivotChart` class dispatches several events to help you follow the query process and provide feedback to the users if needed. For example, you can display a message to the user when the query is invoked and remove it when the query has ended.

```
<mx:Label id="messageToUser"/>
<ilog:PivotChart id="pivotchart" width="100%" height="100%" cube="{cube}"
  invoke="messageToUser.text='Query Invoked'"
  result="messageToUser.text='Query Finished'"
  fault="messageToUser.text='Query Failed'" />
```

You can obviously do any other action you might need to on these listeners. For example, in the fault listener you could call the `invalidateQuery` method to try to re-execute the listener in case the problem comes from a temporary network failure.

Organization Charts

Describes the use of IBM ILOG Elixir organization charts with Adobe® Flex® 3.

In this section

Introduction to organization charts
Describes the uses and layout of organization charts with an example.

Organization chart architecture
Describes the architecture of the organization chart classes.

Creating an organization chart control
Describes an organization chart control and gives an example of the code.

Configuring organization chart data
Describes the facilities for configuring your organization chart data.

Organization chart layout
Describes the layout algorithms for items in organization charts and explains how a layout algorithm is specified.

Organization chart view modes
Describes the two `OrgChart` control view modes, global and local.

Organization chart item renderers
Describes the use of the default item renderer and the detailed item renderer to display the graphic representation of a data item and gives examples of how to set these renderers; describes the use of custom renderers.

Styling organization charts
Describes the properties that you can use to style your organization charts.

Managing organization chart events
Describes the management of events on organization charts.

User interaction with organization charts
Describes the ways that you can interact with an organization chart.

View manipulation API
Describes the methods and properties of the API for manipulating the view of the organization chart.

Adding managed data fields
Explains how to add a managed data field in an organization chart.

Printing organization charts
Explains how to print organization charts.

Introduction to organization charts

An organization chart is often used for corporate intranets to represent employees and their managerial relationships visually.

Such relationships have different representations:

Line
A direct relationship between superior and subordinate.

Lateral
A relationship between different departments on the same hierarchical level.

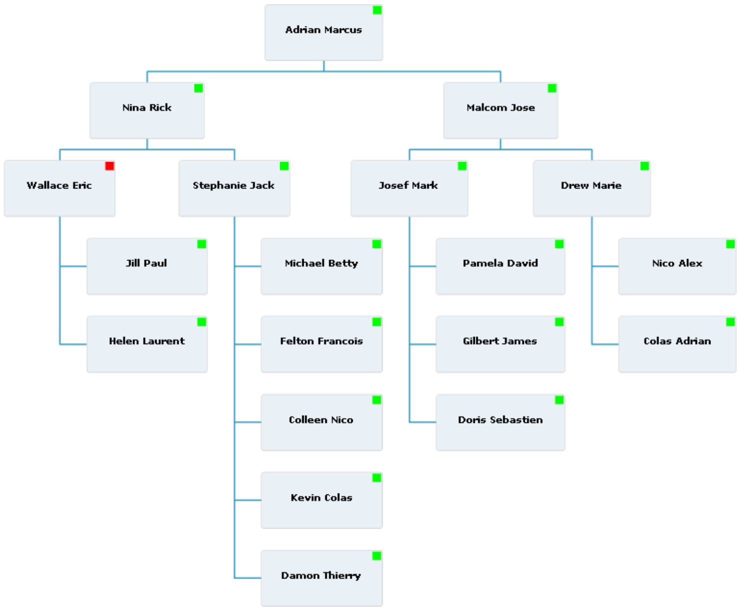
Staff
A relationship between a managerial assistant and other areas. Assistants may be able to offer advice to line managers. However, they have no authority over the actions of line managers.

An IBM ILOG Elixir organization chart can operate in two modes:

Local mode
To facilitate graphical navigation in large employee databases, IBM ILOG Elixir organization charts in local mode display only a given employee, colleague, manager, and subordinate hierarchical level. In this navigation mode, automated drill down is activated to display only meaningful employees in relation to the selected one. For more information, see *Local view mode*.

Global mode
An IBM ILOG Elixir organization chart also provides a global mode where all employees contained in your data sets are displayed at once. For more information, see *Global view mode*.

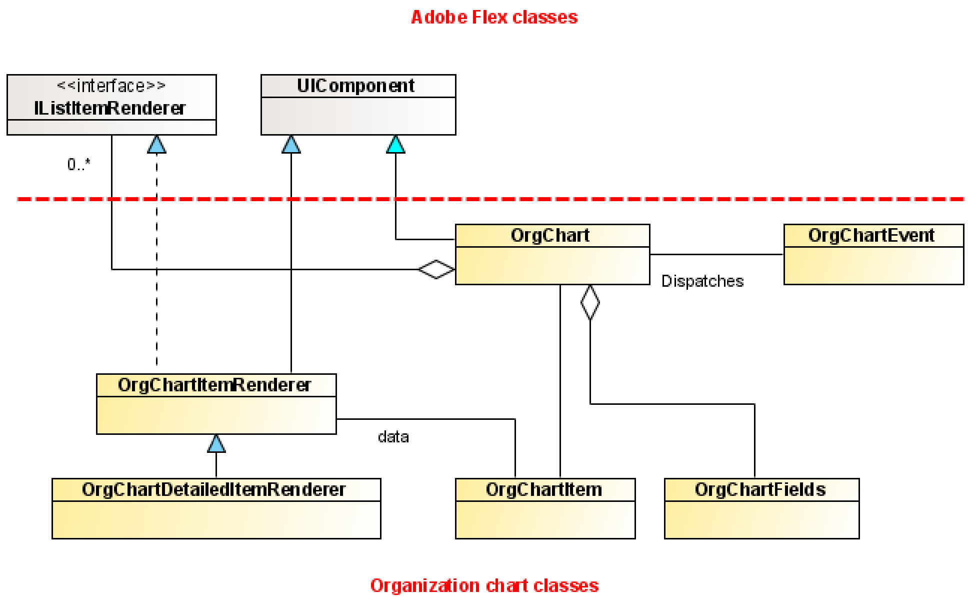
An example of a hierarchical organization chart is shown in the following figure.



In the current version, organization charts are characterized by their layout and the type of presentation box used, see *Organization chart item renderers* for more information.

Organization chart architecture

The organization chart architecture including Adobe® Flex® classes and IBM ILOG Elixir organization chart classes is shown in the following figure.



The organization chart classes shown in the UML diagram are described in the following table.

Main organization chart classes

Organization chart class	Description
`OrgChart`	The main class for organization charts. This class inherits directly from the UIComponent class and is the top level object to be used in MXML.
`OrgChartItemRenderer`	The default item renderer.
`OrgChartDetailedItemRenderer`	An alternative item renderer that displays more data fields.
`OrgChartFields`	This class is the holder of the data field mapping.
`OrgChartItem`	The data set on the item renderer.
`OrgChartEvent`	The type of event fired by a user interaction.

Creating an organization chart control

You can define an organization chart control in MXML using the `<ilog:OrgChart>` tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an `id` value. An organization chart control displays tree data in a specific manner; however, it is similar to the Adobe® Flex® Tree control and can be configured in the same way by using a `dataProvider` as shown in the following code.

```
<ilog:OrgChart width="200" height="400" id="myOrgChart"
   dataProvider="{myDataProvider}" />
```

Configuring organization chart data

Describes the facilities for configuring your organization chart data.

In this section

Organization chart data providers
Describes the possible data providers for organization charts.

Specifying organization chart data with field mappings
Describes the properties for mapping organization chart layout and organization chart item information and how to map them.

Specifying organization chart data with custom functions
Describes how to provide organization chart data with custom functions.

Organization chart data providers

An `OrgChart` control gets its data from a hierarchical data provider that can be one of the following types:

XML
A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` compile-time tag or an XML object. The tags that represent items in the XML data can have any name. The `OrgChart` control reads the XML and builds the display hierarchy based on the nested relationship of the items.

IHierarchicalData
An `IHierarchicalData` or `IHierarchicalCollectionView` implementation instance. These interfaces and their implementations are available with Adobe® Flex® Builder(TM) Professional.

Other objects
An object where the children of this item are contained in a property named `children`.

In addition to specifying the `dataProvider` of the `OrgChart` control, you must also map the different fields required to display the control to fields that exist in the `dataProvider` content, see *Mapping a field to a data provider attribute* for an example.

The object returned by the `dataProvider` property of the `OrgChart` control, regardless of the type of input, is of type `IHierarchicalCollectionView`. When your data can change dynamically (additions, removals, property updates), the best practice is to use a collection, which provides the necessary notifications of changes.

Note: The OrgChart will manipulate the collection to open or close the node, so conflicts will appear if the same collection is used by an `OrgChart` and another component (a `TreeMap` control, for example). The solution is to use different collections and synchronize them.

Specifying organization chart data with field mappings

In the control, there are field properties for mapping organization chart layout and field properties for mapping organization chart item information. You should set the field properties to map fields in the control to the appropriate `dataProvider` fields.

Organization chart layout field properties

The field properties used to map the layout data for `OrgChart` items are described in the following table.

Field properties for organization chart layout

Property	Default value	Purpose
assistantField	assistant	The value of the field whose name is stored in this property indicates whether the item is an assistant of the parent item or not.
layoutField	layout	The value of the field whose name is stored in this property specifies the algorithm used to lay out the child (nonassistant) items.

For more information, see *Organization chart layout*.

Organization chart item renderer field properties

The field properties used to map the data used to render information about the `OrgChart` items are described in the following table.

Field properties for organization chart item information

Property	Default value	Purpose
addressField	address	The value of the field whose name is stored in this property gives the address of the person (personal or professional).
businessUnitField	businessUnit	The value of the field whose name is stored in this property gives the business unit of the person.
emailField	email	The value of the field whose name is stored in this property gives the email of the person.
faxField	null	The value of the field whose name is stored in this property gives the fax number of the person.
instantMessagerField	instantMessager	The value of the field whose name is stored in this property gives the instant messager identifier of this person.
locationField	location	The value of the field whose name is stored in this property gives the location of the person (country, region, site...).
nameField	name	The value of the field whose name is stored in this property gives the name of the person.
mobilePhoneField	mobilePhone	The value of the field whose name is stored in this property gives the mobile phone number of the person.
pictureField	picture	The value of the field whose name is stored in this property gives the source of the picture of the person.
phoneField	phone	The value of the field whose name is stored in this property gives the phone number of the person.
positionField	position	The value of the field whose name is stored in this property gives the position of the person, for example, manager, software engineer...
presenceIndicatorField	presenceIndicator	The value of the field whose name is stored in this property indicates whether the person is present, away, or travelling.
timeZoneField	null	The value of the field whose name is stored in this property gives the time zone of the person.

Mapping a field to a data provider attribute

The following code shows the field property `nameField` mapping the `name` field in the control to the `dataProvider` attribute `label`.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
   xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
```

```
  <ilog:OrgChart width="400" height="200" >
    <mx:XML>
      <employee label="Firstname Name" />
    </mx:XML>
    <ilog:fields>
      <ilog:OrgChartFields nameField="label" />
    </ilog:fields>
  </ilog:OrgChart>
</mx:Application>
```

Specifying organization chart data with custom functions

If the data values for a field in an organization chart control are not directly available in one of the fields of the data provider items, you can specify a custom function to provide the `OrgChart` control with the required information.

If a custom function and a field are both specified, the function takes precedence over the field. The function properties used to specify custom functions are listed in the following table.

Functions and their corresponding field names

Function property	Corresponding field name
addressFunction	address
assistantFunction	assistant
businessUnitFunction	businessUnit
emailFunction	email
faxFunction	fax
instantMessagerFunction	instantMessager
layoutFunction	layout
locationFunction	location
mobilePhoneFunction	mobilePhone
nameFunction	name
pictureFunction	picture
phoneFunction	phone
positionFunction	position
presenceIndicatorFunction	presenceIndicator
timeZoneFunction	timeZone

To illustrate the use of custom functions, the following code shows how a custom function uses one value read from the data item in the `dataProvider` to retrieve another data item value.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      private function getAddress(item:Object):String {
        var address:String;
        switch(item.location) {
          case "location 1":
            address = "address 1";
            break
          case "location 2":
            address = "address 2";
            break
          default:
            address = "main address";
        }
        return address;
      }
    ]]>
  </mx:Script>
  <ilog:OrgChart width="400" height="200" >
    <ilog:fields>
      <ilog:OrgChartFields addressFunction="{getAddress}" />
    </ilog:fields>
  </ilog:OrgChart>
</mx:Application>
```

If no custom function is set for a field and the field is set to `null`, then the field is hidden. This is particularly useful if the item renderer displays data that is by default not available in the `dataProvider`.

Organization chart layout

Describes the layout algorithms for items in organization charts and explains how a layout algorithm is specified.

In this section

Standard
Describes the standard layout, which is the default.

Right hanging
Describes the right hanging layout.

Left hanging
Describes the left hanging layout.

Both hanging
Describes the both hanging layout.

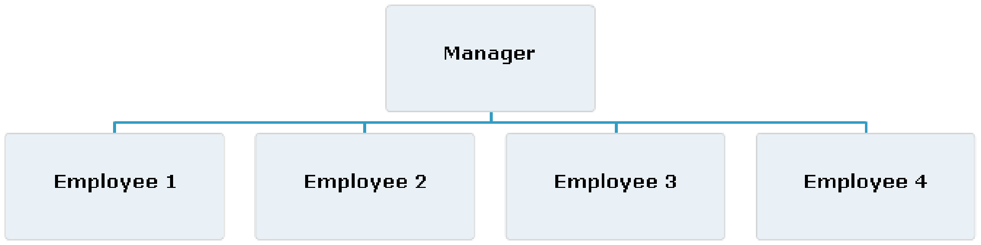
Assistants
Describes the specific treatment for assistants.

Specifying the layout algorithm
Describes the different ways of specifying a layout algorithm.

Standard

The default layout is standard. The value of the `layout` field is `standard`.

The following figure shows an example of `standard` layout.



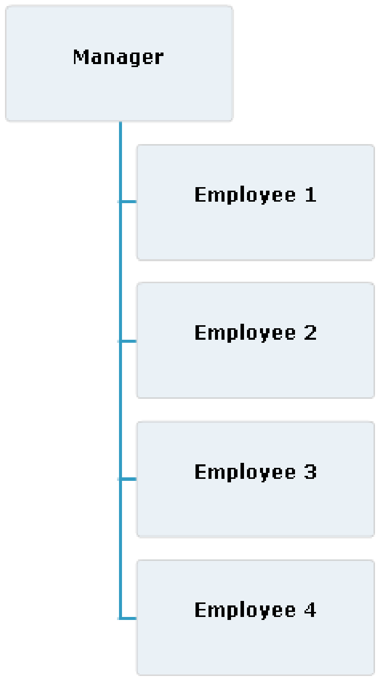
The children of a data provider item can be laid out differently by the use of a different algorithms: right hanging, left hanging, or both hanging.

All algorithms allow you to mix different layouts in the same organization chart. There is also specific treatment available for assistants.

Right hanging

This layout distributes all the children in a right hanging column. The value of the `layout` field is `rightHanging`.

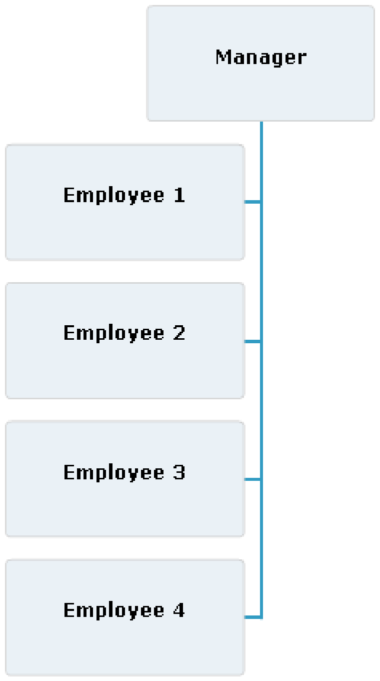
The following figure shows an example of `rightHanging` layout.



Left hanging

This layout distributes all the children in a left hanging column. The value of the `layout` field is `leftHanging`.

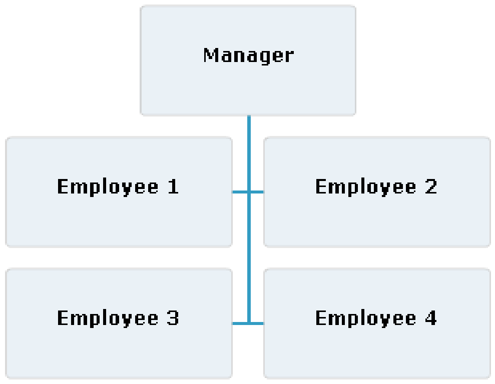
The following figure shows an example of `leftHanging` layout:



Both hanging

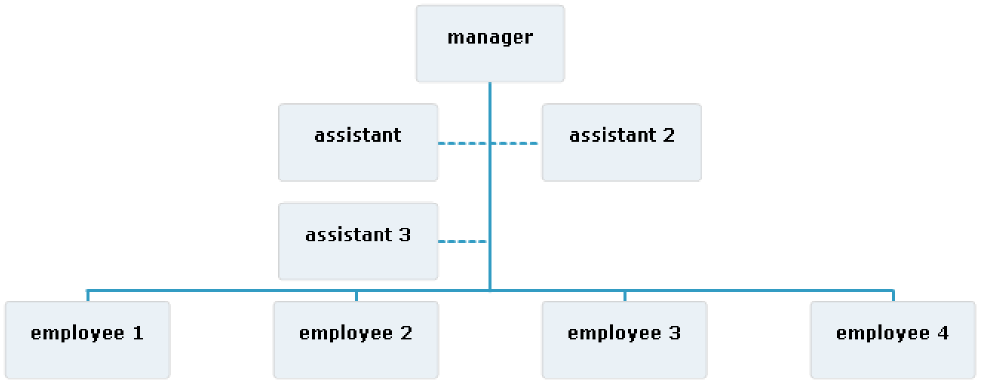
This layout distributes the children equally in columns and rows. The value of the `layout` field is `bothHanging`.

The following figure shows an example of `bothHanging` layout:



Assistants

Assistants are children of a `dataProvider` item that have a different relationship with the parent node. They are laid out in a dedicated part of the tree using the `bothHanging` layout algorithm.



To specify that a data item is an assistant of its parent, the field of this data item that contains the assistant value must be set to `true`.

Specifying the layout algorithm

The algorithm used to lay out the children of a given item is specified in their parent. The following example shows how to specify the algorithm using the `bothHanging` value for the `layout` property.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:OrgChart width="400" height="200" >
    <mx:XML>
      <employee name="person 1" position="manager" layout="bothHanging">
        <employee name="person 2" position="employee" />
        <employee name="person 3" position="employee" />
      </employee>
    </mx:XML>
  </ilog:OrgChart>
</mx:Application>
```

The following example shows how to use a function to deduce the layout of an item.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      private function getItemLayout(item:Object):String {
        return item.@position == "manager" ? "bothHanging" : null;
      }
    ]]>
  </mx:Script>
  <ilog:OrgChart width="400" height="200" >
    <mx:XML>
      <employee name="person 1" position="manager">
        <employee name="person 2" position="employee" />
        <employee name="person 3" position="employee" />
      </employee>
    </mx:XML>
  </ilog:OrgChart>
</mx:Application>
```

Organization chart view modes

Describes the two `OrgChart` control view modes, global and local.

In this section

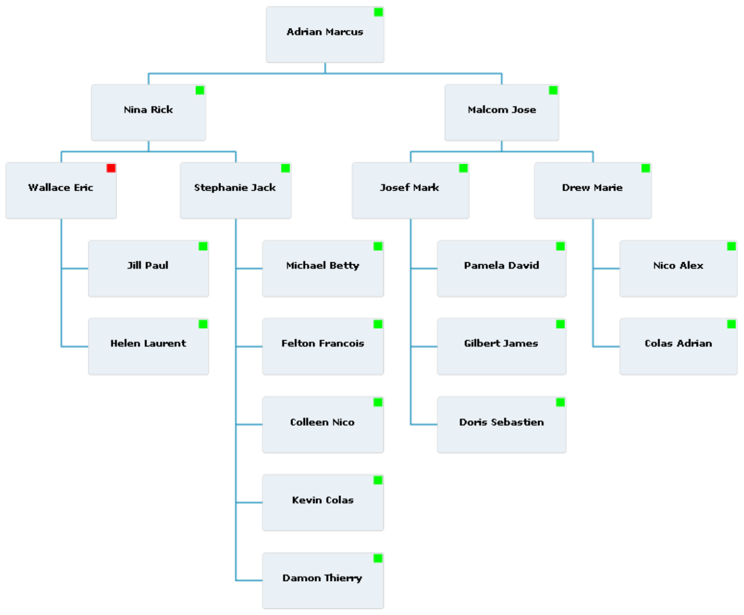
Global view mode
Describes the use of the global view mode, which is the default.

Local view mode
Describes the use of the local view mode.

Global view mode

This is the default view mode. All the items of the `dataProvider` are represented with no filtering.

An example of global view mode is shown in the following figure.



Note that to make the initial view display all the items, you must set the `initialPosition` style property of the organization chart to `fitToContents`.

Local view mode

The purpose of the local view mode is to display the local hierarchy of a specific item known as the local item. This facilitates navigation through large datasets. The following items are displayed:

- The local item.
- The siblings of the local item.
- At the most `n` levels of local item parents (by default 1). `n` is specified by the `upperLevelLimit` property.
- At the most `m` levels of local item children (by default 1). `m` is specified by the `lowerLevelLimit` property.

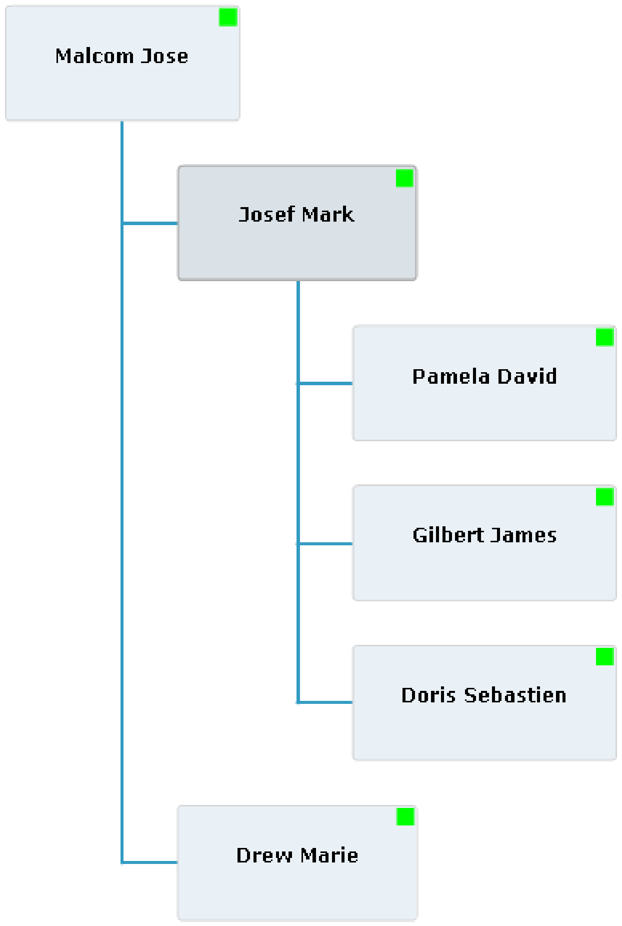
By default the local item is the root of the hierarchy.

The local item can be changed by highlighting the item and clicking the icon or by pressing the **Enter** key.

To change the local item programatically, specify an item of the `dataProvider` in the `localItem` property.

If the default item renderers are used, the local item is visually identified by a thicker border and darker color.

The following figure displays the same `dataProvider` as given in *Global view mode*, but in local view mode. The local item is `Josef Mark` and the upper and lower limits are set to 1.


Malcom Jose
Josef Mark
Pamela David
Gilbert James
Doris Sebastien
Drew Marie

Organization chart item renderers

Describes the use of the default item renderer and the detailed item renderer to display the graphic representation of a data item and gives examples of how to set these renderers; describes the use of custom renderers.

In this section

Default item renderer
Describes the use of the default item renderer to display a data item.

Detailed item renderer
Describes the use of the detailed item renderer to display a data item.

Setting the item renderer
Gives some examples of item renderers.

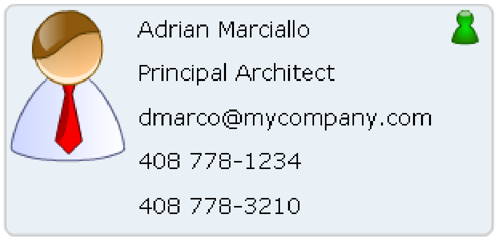
Custom item renderers
Describes the use of custom item renderers to display data items.

Default item renderer

The default item renderer displays a box with the following fields:

- ♦ Picture
- ♦ Presence indicator
- ♦ First name
- ♦ Last name
- ♦ Position
- ♦ E-mail
- ♦ Phone
- ♦ Mobile phone

The following figure shows an example of the default item renderer:


Adrian Marciallo

Principal Architect

dmarco@mycompany.com

408 778-1234

408 778-3210

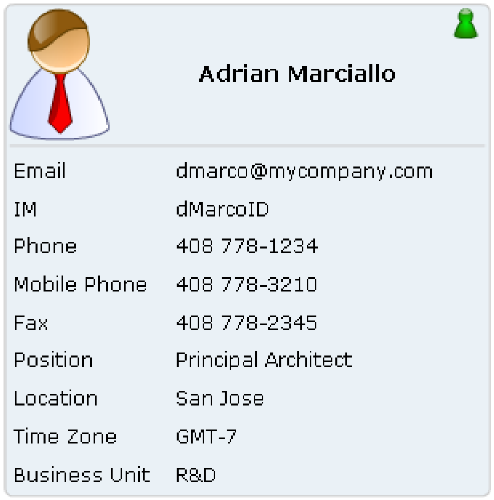
This item renderer is defined by the `OrgChartItemRenderer` class.

Detailed item renderer

The detailed item renderer displays a box with more fields than the default item renderer. The fields are:

- Picture
- Presence indicator
- First name
- Last name
- Email
- Instant messager ID
- Phone
- Mobile phone
- Fax
- Position
- Office location
- Address as a tool tip of the location label
- Time zone
- Business unit

The following figure shows an example of the detailed item renderer:



Adrian Marciallo

Email	dmarco@mycompany.com
IM	dMarcoID
Phone	408 778-1234
Mobile Phone	408 778-3210
Fax	408 778-2345
Position	Principal Architect
Location	San Jose
Time Zone	GMT-7
Business Unit	R&D

This item renderer is defined by the `OrgChartDetailedItemRenderer` class.

Note: The `timeZoneField` and `faxField` organization chart properties are `null` by default and therefore not displayed. For more information, see *Organization chart item renderer field properties*.

Setting the item renderer

The following example shows how to set an item renderer in MXML.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:OrgChart width="400" height="200" >
    <ilog:itemRenderer>
      <mx:Component>
        <ilog:OrgChartDetailedItemRenderer/>
      </mx:Component>
    </ilog:itemRenderer>
  </ilog:OrgChart>
</mx:Application>
```

The following example shows an item renderer set programatically.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
    <![CDATA[
      import ilog.orgchart.OrgChartDetailedItemRenderer;

      private function setDetailedItemRenderer(event:Event):void {
        chart.itemRenderer = new ClassFactory(OrgChartDetailedItemRenderer);
      }
    ]]>
  </mx:Script>
  <ilog:OrgChart id="chart" width="400" height="200" />
  <mx:Button click="setDetailedItemRenderer(event)" />
</mx:Application>
```

Custom item renderers

An `OrgChart` control can use custom renderers to display the data items. A custom item renderer must:

- ♦ Extend a `UIComponent` subclass.
- ♦ Implement the `IListItemRenderer` interface.
- ♦ Return the same height for every data provider item.

Before you decide whether and how to develop a custom item renderer, make sure you read about *Custom Component Development* in the Adobe® Flex® documentation.

The contents of the `data` property for an item renderer are of the type `OrgChartItem`. This type gives access to the data fields to display in the item renderer.

The following example shows how to declare an inline custom item renderer that displays a label in a box and whose border color depends on the location of the person.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:OrgChart width="400" height="200"  dataProvider="{xml}">
    <ilog:itemRenderer>
      <mx:Component>
        <mx:VBox height="30" backgroundColor="{getBorderColor(data)}">
          <mx:Script>
            <![CDATA[
              private function getBorderColor(data:Object):uint {
                var color:uint;
                switch(data.location) {
                  case "location 1":
                    color = 0xeeeeee;
                    break;
                  case "location 2":
                    color = 0xcccccc;
                    break;
                  default:
                    color = 0x333333;
                }
                return color;
              }
            ]]>
          </mx:Script>
          <mx:Label text="{data.name}" />
        </mx:VBox>
      </mx:Component>
    </ilog:itemRenderer>
  </ilog:OrgChart>
</mx:Application>
```

The `OrgChart` layout imposes constraints on the item renderers which affect how they are sized.

These constraints are as follows:

- ♦ An `OrgChart` item renderer is constrained to a fixed height for all the data it represents.
- ♦ The `OrgChart` measures the item renderer that represents the root data item and assigns its height to all the item renderers.
- ♦ The size of an item renderer is the same at every level of detail. When you zoom in on an `OrgChart` control, the item renderers are not resized, they only show or hide subcomponents.
- ♦ The size of each item renderer is computed according to the layout for the data item concerned and cannot be changed.
- ♦ The resulting width of an item renderer is the width of the widest item renderer in the same branch.

If, in the following example of three nodes, the item renderers that represent p1, p2 and p3 are 100, 200, and 100 pixels wide respectively, the width of all the item renderers will be set to 200 pixels.

```
<p name="root">
    <p name="p1"/>
    <p name="p2"/>
    <p name="p3"/>
```

To summarize, an item renderer must return a correct measured width and a constant height.

Styling organization charts

Describes the properties that you can use to style your organization charts.

In this section

Layout properties
Describes the layout properties that affect items.

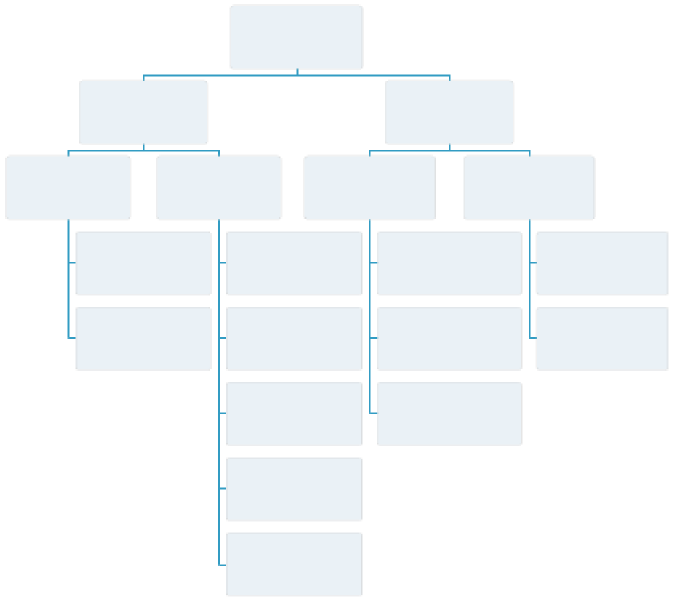
Link properties
Describes the style properties that links may have.

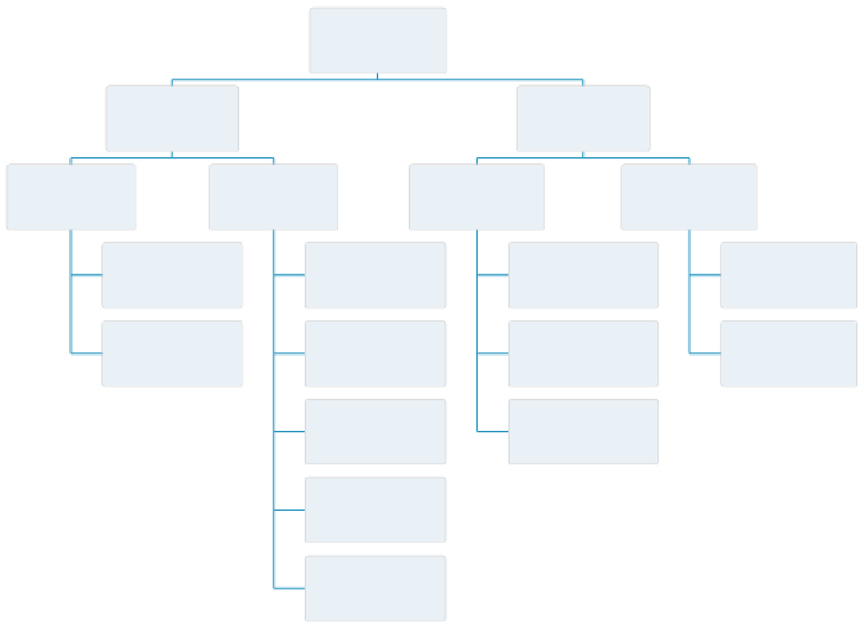
Item properties
Describes the style properties that items may have in different states and explains how to set these properties.

Layout properties

The layout can be configured by specifying the `layoutXPadding` and `layoutYPadding` style properties. These properties define the minimum distance between items on the respective axes.

The following figure shows two examples: in the first, the minimum distance between items is small; in the second, the minimum distance is set to the default value (there is more space between the items).

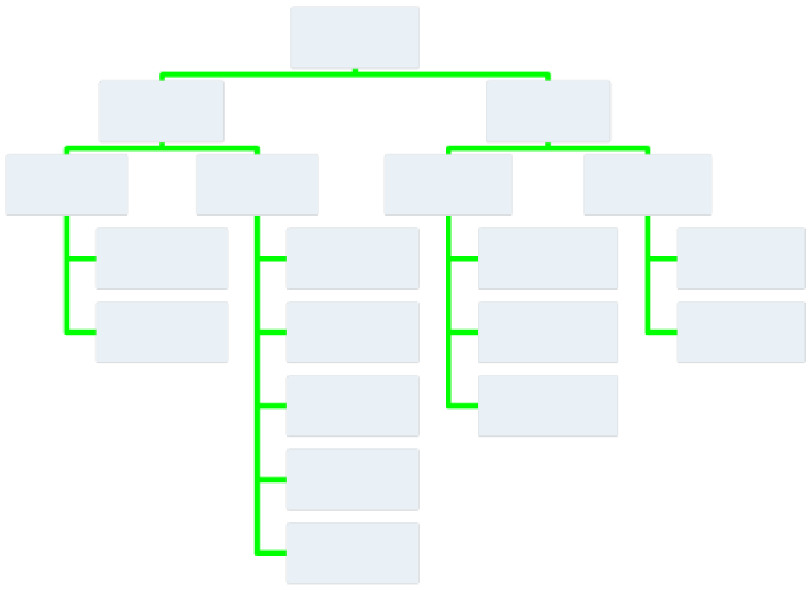




Link properties

The color and thickness of links that connect the items can be configured by setting the `linkColor` and `linkThickness` style properties respectively.

The following figure shows an example of an organization chart with the link color set to green and the link thickness set to 3 pixels.



Item properties

Items are displayed by an item renderer so the supported style properties depend on the item renderer used. The item renderers provided with the `OrgChart` controls support multiple states (highlighted, selected, and so on), therefore different style properties are exposed. The available states and the corresponding style properties are described in the following table.

Organization chart item states and style properties

State	Specific style properties
Regular	`backgroundColor`, `borderColor`, `color`
Selected	`selectedBackgroundColor`, `selectedBorderColor`, `selectedTextColor`
Highlighted	`rollOverBackgroundColor`, `rollOverBorderColor`, `rollOverTextColor`
Selected and highlighted	`selectedRollOverBackgroundColor`, `selectedRollOverBorderColor`, `selectedRollOverTextColor`

The complete list of style properties is available in the *IBM ILOG Elixir Language Reference*, see the `OrgChartItemRenderer` class.

The item renderer `styleName` is set to the `itemStyleName` style property value. This means that by default, the item renderer inherits the style specified by the `itemStyleName` style property.

The following example shows how to set the style by using the `itemStyleName` style property.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
 <mx:Style>
   .myItemStyle {
     backgroundColor: #ff0000;
     rollOverBackgroundColor:#00ff00;
     selectedBackgroundColor: #0000ff;
   }
 </mx:Style>
 <ilog:OrgChart id="chart" width="400" height="200"
    itemStyleName="myItemStyle"/>
</mx:Application>
```

Alternatively you can set the style of the `OrgChartItemRenderer` in MXML directly as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <ilog:OrgChart width="400" height="200"  dataProvider="{xml}">
```

```
    <ilog:itemRenderer>
      <mx:Component>
        <ilog:OrgChartItemRenderer backgroundColor="#ff0000"
          rollOverBackgroundColor="#00ff00"
          selectedBackgroundColor="#00ff00" />
      </mx:Component>
    </ilog:itemRenderer>
  </ilog:OrgChart>
</mx:Application>
```

Managing organization chart events

An `OrgChart` control can fire several kinds of specific event in response to user interactions. You can create an event handler for each of these events.

For example, if you register a listener for an `itemClick` event on an `OrgChart` class, this listener is called every time an item is clicked. The listener can then access the data for this item.

You can listen for the following `OrgChart` data events.

Organization chart data events

Organization chart data event	Description
`change`	Indicates that the selection changed as a result of user interaction.
`itemClick`	Indicates that the user clicked the pointer over a visual item in the control.
`itemDoubleClick`	Indicates that the user double-clicked the pointer over a visual item in the control.
`itemRollOut`	Indicates that the user rolled the pointer out of a visual item in the control.
`itemRollOver`	Indicates that the user rolled the pointer over a visual item in the control.

The `OrgChart` events are of the `OrgChartEvent` type.

The following code shows an interaction when an item is clicked.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex" >
  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import ilog.orgchart.OrgChartEvent;
      private function displayItemName(event:OrgChartEvent):void {
        Alert.show("You have clicked on " + event.item.name, "Click Event");
      }
    ]]>
  </mx:Script>
  <ilog:OrgChart itemClick="displayItemName(event)" />
</mx:Application>
```

User interaction with organization charts

For more information, see *Handling user interactions with charts* and *Selecting chart items* in the *Flex 3 Developer's Guide*.

Selection

When an item is clicked, it is selected and an `itemClick` event is fired. To enable multiple selection, set the `allowMultipleSelection` property to `true`.

The current selection is available in the `selectedItems` property of the `OrgChart`. You can disable selection by setting the `allowSelection` property to `false`.

The selection actions are shown in the following table.

Selection actions

Action	Mouse	Keyboard
Select	Click an item in the organization chart.	Highlight an item and press Space.
Extend a selection	Hold down the CTRL key and click the items.	Highlight an item and press CTRL + Space.
Clear a selection	Click anywhere on the chart background.	--
Highlight	Move the pointer over an item.	--
Select a neighboring item.	--	CTRL + Arrow keys.

Navigation

An `OrgChart` view can be zoomed and panned to display a specific part of the chart. The available actions are described in the following table:

Navigation actions

Action	Mouse	Keyboard
Zoom in	Rotate the mouse wheel forward.	+
Zoom out	Rotate the mouse wheel backward.	-
Pan	Click and drag.	Arrow keys

Navigation can be disabled by setting the `allowNavigation` property to `false`.

In addition to these actions, an `OrgChart` control provides an API for navigation purposes. See *View manipulation API*.

An `OrgChart` control has the following zoom constraints:

♦ The minimum zoom level displays the entire chart. The minimum zoom level can be specified by setting the `minZoomLevel` property

- The maximum zoom level displays an item. The maximum can be specified by setting the `maxZoomLevel` property.

The initial view position is determined by the `initialPosition` style property.

Valid values of `initialPosition` are:

- `centeredOnRoot`: This is the default value. The view is centered on the root of the organization chart and not scaled.
- `fitToContents`: The view is scaled to display the entire organization chart.

View manipulation API

The specific methods and properties for navigation are shown in the following table.

Using the API methods and properties for navigation

Method/Property	Description
`centerOn()`	Centers the view on the specified item (without changing the zoom level).
`fitToContents()`	Shows the entire chart.
`fitToSubTree()`	Zooms to show the entire tree branch.
`scale`	Allows you to set the zoom level.
`zoomBy()`	Zooms the view by the specified factor.

The methods in this table have an `animate` parameter to display smooth view transitions.

Since IBM ILOG Elixir 2.0, it is possible to manipulate the coordinates of the visible area of the `OrgChart` component through the API.

The event `OrgChartEvent.LAYOUT_CHANGE` is dispatched when the OrgChart has computed the layout of the item of the data provider.

Once the layout is computed, you have access to the following:

♦ The `modelBounds` property, which contains the bounds of the chart as a whole

♦ The `getItemBounds(item)` function, which returns the bounds of the item renderer associated with the specified item

The event `OrgChartEvent.VISIBLE_LAYOUT_CHANGE` event is dispatched when the visible area has changed. The `visibleBounds` property gives access to the visible area coordinates.

Adding managed data fields

The organization chart control manages a list of fields displayed by the default item renderers.

To add fields and display them using the default item renderers, you need to subclass the main classes.

The following example adds a managed salary field.

To add a managed salary field:

1. Subclass the `OrgChartFields` class that holds the field data mapping

```
package
{
  import ilog.orgchart.OrgChartFields;
  import flash.events.Event;

  /**
   * This sample class shows how to add field mappings.
   */
  public class ExtendedFields extends OrgChartFields
  {
    private var _salaryField:String = "salary";

    /**
     * @private
     */
    public function set salaryField(value:String):void {
      if (_salaryField != value) {
        _salaryField = value;
        dispatchEvent(new Event(Event.CHANGE));
      }
    }

    [Inspectable (category="OrgChart", defaultValue="salary")]
    /**
     * The name of the property chosen to determine the salary
     * of a person, if the <code>salaryFunction</code> is not set.
     *
     * @default "salary"
     */
    public function get salaryField():String {
      return _salaryField;
    }

    private var _salaryFunction:Function = null;

    /**
     * @private
     */
    public function set salaryFunction(value:Function):void {
      if (_salaryFunction != value) {
        _salaryFunction = value;
        dispatchEvent(new Event(Event.CHANGE));
      }
    }

    /**
     * The function used to determine the salary of a person.
     * It must have the following signature:
     * <pre>
     * salaryFunction(<i>item:Object</i>):String
     * </pre>
     * This returns the salary from the
     * <code>item</code> provided by the data provider.
     *
     * @default null
     */
    public function get salaryFunction():Function {
      return _salaryFunction;
    }
  }
}
```

2. Subclass the `OrgChartItem` class to provide the item renderers with access to the data.

```
package
{
  import ilog.orgchart.OrgChartItem;
  import mx.core.IUIComponent;

  /**
   * This sample data item shows how to add a data field to the
   * <code>OrgChartItem</code>.
   */
  public class ExtendedDataItem extends OrgChartItem
  {
    public function ExtendedDataItem(owner:IUIComponent, item:Object,
                                     data:Object) {
      super(owner, item, data);
    }

    public function get salary():String {
      var fields:ExtendedFields = orgChart.fields as ExtendedFields;
      return getFieldValue(fields.salaryField, null,
                           fields.salaryFunction) as String;
    }
   }
 }
```

3. Subclass the `OrgChart` class to create `ExtendedDataItem` instances and pass them to the item renderers.

```
package
{
  import ilog.orgchart.OrgChart;
  import ilog.orgchart.OrgChartItem;

  /**
   * This sample OrgChart shows how to use an OrgChartItem subclass.
   */
  public class ExtendedOrgChart extends OrgChart
  {
    override protected function createOrgChartItem(item:Object)
:OrgChartItem {
      var collection:IHierarchicalCollectionView = dataProvider as

IHierarchicalCollectionView;
      var hData:IHierarchicalData = collection.source;
      return new ExtendedDataItem(this, item, hData.getData(item));
    }
  }
 }
```

4. Subclass the two default item renderers to display the new field.

   The following example shows the subclass of `OrgChartItemRenderer`.

```
package
{
  import ilog.orgchart.OrgChartItemRenderer;
  import ilog.orgchart.OrgChartItem;
  import mx.core.FlexTextField;

 /**
  * This sample item renderer shows how to add a field label to the
  * OrgChartItemRenderer
  */
  public class ExtendedItemRenderer extends OrgChartItemRenderer
  {

    private var _salaryLabel:FlexTextField;

    override protected function getFieldLabels():Array {
      var labels:Array = super.getFieldLabels();
      labels.push(_salaryLabel);
      return labels;
    }

    override protected function applyData(data:OrgChartItem):void {
      _salaryLabel = setLabelProperty(_salaryLabel, "salary");
      super.applyData(data);
    }
  }
```

   The following example shows the subclass of `OrgChartDetailedItemRenderer`.

```
package
{
  import ilog.orgchart.OrgChartDetailedItemRenderer;
  import mx.core.FlexTextField;
  import ilog.orgchart.OrgChartItem;

  /**
   * This sample item renderer shows how to add a field label to the
   * OrgChartDetailedItemRenderer
   */
  public class ExtendedDetailedItemRenderer extends
OrgChartDetailedItemRenderer
  {
    private var _salaryLabel:FlexTextField;
    private var _salaryLeftLabel:FlexTextField;

    override protected function createChildren():void {
      _salaryLeftLabel = new FlexTextField();
      addChild(_salaryLeftLabel);
      super.createChildren();
    }

    override protected function setLeftLabelsText():void {
      super.setLeftLabelsText();
      if (_salaryLeftLabel != null) {
        _salaryLeftLabel.text = "Salary";
      }
    }

    override protected function getFieldLabels():Array {
      var labels:Array = super.getFieldLabels();
      labels.push(_salaryLabel);
      return labels;
    }

    override protected function getFieldValueLabels():Array {
      var labels:Array = super.getFieldValueLabels();
      labels.push(_salaryLeftLabel);
      return labels;
    }

    override protected function applyData(data:OrgChartItem):void {
      _salaryLabel = setLabelProperty(_salaryLabel, "salary");
      super.applyData(data);

    }
  }
}
```

The following MXML sample shows how to use all the components.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
                xmlns:local="*"
                layout="absolute">
```

```
  <local:ExtendedOrgChart id="chart" width="100%" height="100%" >
    <mx:XML>
      <employee name="Employee 1" email="e1@company.com" salary="123456" >
        <employee name="Employee 2" email="e1@company.com" salary="123456" />

        <employee name="Employee 3" email="e1@company.com" salary="123456" />

      </employee>
    </mx:XML>
    <local:fields>
      <local:ExtendedFields />
    </local:fields>
    <local:itemRenderer>
      <mx:Component>
        <local:ExtendedItemRenderer />
      </mx:Component>
    </local:itemRenderer>
  </local:ExtendedOrgChart>

</mx:Application>
```

Printing organization charts

Explains how to print organization charts.

In this section

Printing organization charts on multiple pages
Explains how to print large organization charts on more than one page

Adjusting the print layout
Explains how to set print layout properties.

Sizing a printed organization chart
Explains how to set properties to size the printed organization chart.

Example of printing with multipage controls
Gives a code example showing printing with a multipage `PrintOrgChart` control.

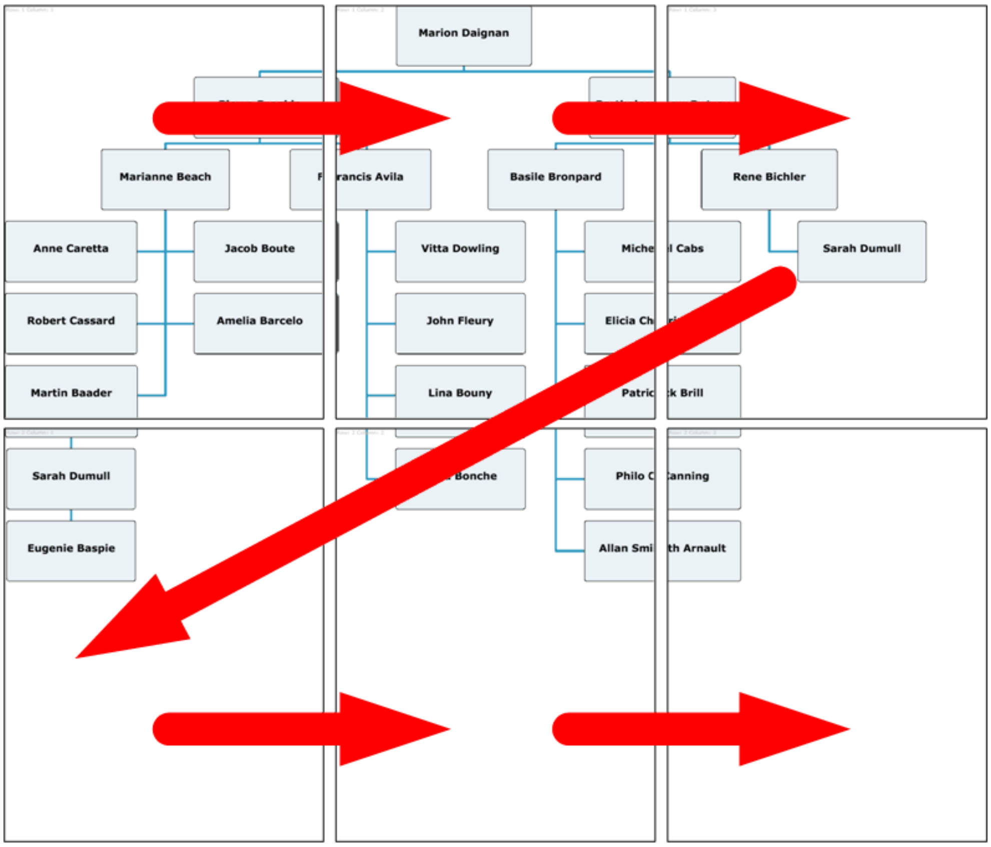
Printing organization charts on multiple pages

The `PrintOrgChart` class allows the printing of large organization charts on more than one page

The process of printing a large organization chart is similar to printing a multipage data grid with the Adobe® Flex® 3 `PrintDataGrid` class. See *Printing multipage output* in *Advanced Flex Programming>Printing* in the Adobe Flex 3 documentation. The main difference is that pages represent a matrix of rows and columns. To visualize your organization chart, you have to reassemble the pages.

The `validNextPage` property is `true` if the `PrintOrgChart` control has data beyond the current print page. You use this property to determine whether you need to format and print an additional page.

The `nextPage()` method moves to the next set of data to print. Pages are printed row by row, as shown in the following figure.



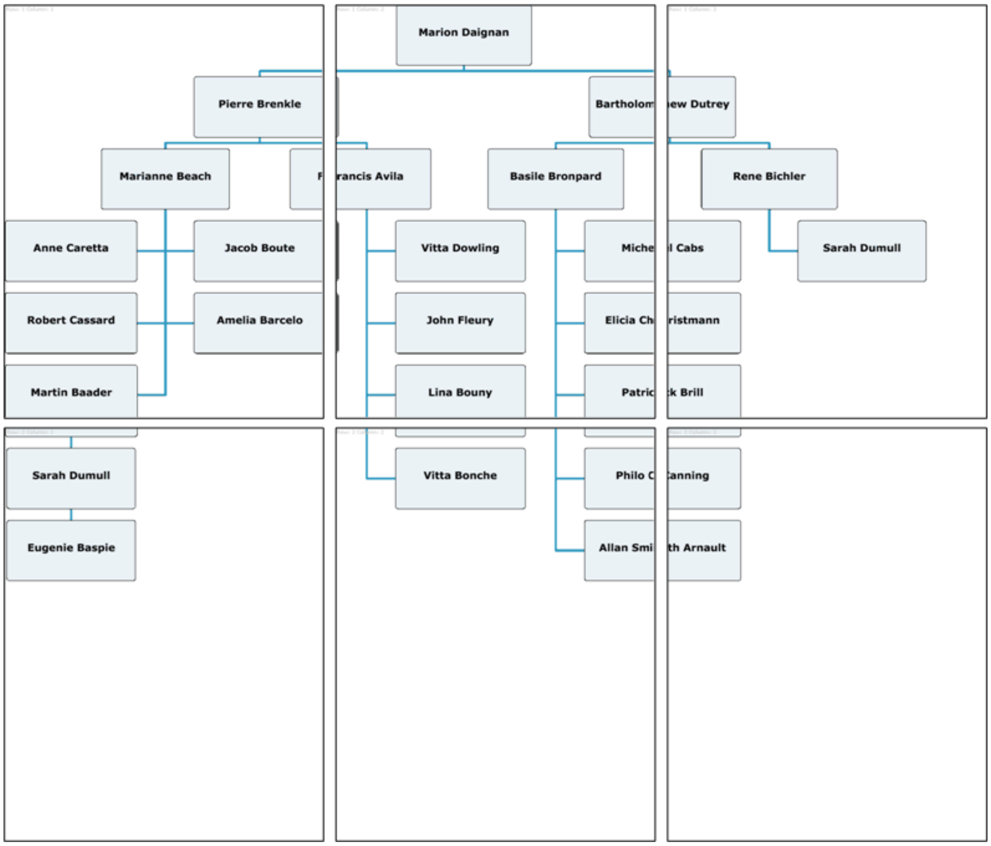
The following code shows a loop that prints an organization chart on multiple pages.

```
// Queue the first page
printJob.addObject(thePrintView);
// While there are more pages, print them
while (thePrintView.myOrgChart.validNextPage) {
    //Display next set of data (page)
    thePrintView. myOrgChart.nextPage();
    //Queue the additional page.
    printJob.addObject(thePrintView);
}
```

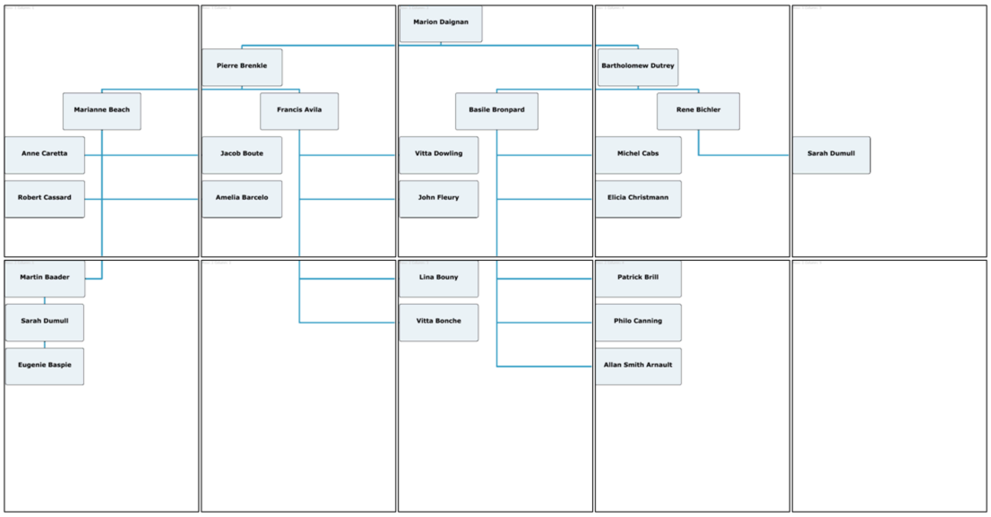
Adjusting the print layout

You can choose whether to allow employee boxes to be cut by page breaks. If you set `allowBoxCut` to `false`, this can result in a greater number of pages.

The following figure shows an organization chart printed with `allowBoxCut` set to `true`.



The following figure shows the same organization chart printed with allowBoxCut = false. The number of pages is greater: 10 pages, compared with 6 when `allowBoxCut` is set to `true`.


Marion Daignan
Pierre Brenkle
Bartholomew Dutrey
Marianne Beach
Francis Avila
Basile Bronpard
Rene Bichler
Anne Caretta
Jacob Boute
Vitta Dowling
Michel Cabs
Sarah Dumull
Robert Cassard
Amelia Barcelo
John Fleury
Elicia Christmann
Martin Baader
Lina Bouny
Patrick Brill
Sarah Dumull
Vitta Bonche
Philo Canning
Eugenie Baspie
Allan Smith Arnault

Sizing a printed organization chart

The `printingScalingFactor` property allows you to change the size of the printed chart. By default this property is set to `1.0`, which means the printing area will have a size of `modelWidth` by `modelHeight`. The example settings are as follows: size 500x300, to be printed on pages of size 300x500, with `allowBoxCut` set to `true`.

To print an organization chart with the example property settings :

1. Set `printingScalingMode` to `1.0`.

   You will need 2 pages (1 row of 2 pages) to print your chart.

2. Set `printingScalingFactor` to `2.0`.

   Your chart will have a size of (2*`modelWidth` x 2*`modelHeight`) = (1000 x 600).

You will need 4 rows and 2 columns of pages for a total of 8 pages to print your chart.

Example of printing with multipage controls

The following example prints an organization chart on multiple pages. It also shows how you can put information on top pages and left pages.

This example uses the technique of selectively showing and hiding the top and left labels, depending on the page being printed.

The application consists of the following files:

The application file
This file displays the chart to the user with a **Print** button. The file includes the code to initialize the view, to get the data, and to handle the print request. It uses the `FormPrintView` MXML component as a template for the printed output.

The print output component file
The print output component file, `FormPrintView.mxml`, formats the printed output.

`FormPrintView.mxml` has two major elements:

- ♦ The print output template, which includes the `PrintOrgChart` control and uses two labels for left and top pages.
- ♦ The `showLabel()` function, which determines which sections of the template to include in a particular page of the output, based on the page type: top or left.

This example also adds a watermark on pages to print the relative position of each page. For example, the first page shows: `Row: 1 - Column: 1`.

The following code shows the application file for the multipage printing example.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:ilog="http://
www.ilog.com/2007/ilog/flex" xmlns:printing="ilog.printing.*"
  backgroundColor="0xFFFFFF" backgroundGradientAlphas="[]">

  <mx:Script>
    <![CDATA[
  import mx.containers.VBox;
  import mx.controls.Label;
  import mx.containers.Canvas;
  import mx.printing.FlexPrintJobScaleType;
  import mx.printing.FlexPrintJob;

  // The function to print the output.
  public function doPrint():void
  {
    // Create a FlexPrintJob instance.
    var printJob:FlexPrintJob = new FlexPrintJob();
    printJob.printAsBitmap = false;
    // Start the print job.
    if (printJob.start() == false)
      return;
    // Create a FormPrintView control
    // as a child of the application.
    var thePrintView:FormPrintView = new FormPrintView();
    addChild(thePrintView);

    // Set the data provider of the FormPrintView
    // component's DataGrid to be the data provider of
    // the displayed DataGrid.
    thePrintView.chart.dataProvider = chart.dataProvider;

    // Set the print view properties.
    thePrintView.width=printJob.pageWidth;
    thePrintView.height=printJob.pageHeight;
    thePrintView.validateNow();

    // Queue first page
    printJob.addObject(thePrintView);

    while (thePrintView.chart.validNextPage)
    {
      thePrintView.chart.nextPage();
      thePrintView.row = thePrintView.chart.currentRow + 1;
      thePrintView.column = thePrintView.chart.currentColumn + 1;

      // Relayout the print view
      if (thePrintView.chart.currentRow == 0)
        thePrintView.showLabel(thePrintView.topLabel, true)
      else
        thePrintView.showLabel(thePrintView.topLabel, false)

      if (thePrintView.chart.currentColumn == 0)
        thePrintView.showLabel(thePrintView.leftLabel, true)
      else
        thePrintView.showLabel(thePrintView.leftLabel, false)

      // Queue the additional page.
      printJob.addObject(thePrintView);
    }

    // All pages are queued; remove the FormPrintView
    // control to free memory.
    removeChild(thePrintView);
    // Send the job to the printer.
    printJob.send();
  }

  [Bindable]
  private var xml:XML =
     <person layout="standard" name="Marion Daignan" email="mdaignan@mycompany.
com" phone="4155550012" mobilePhone="4155550013" fax="4155550014"
instantMessager="mdaignan" position="PrincipalArchitect" location="San
Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="sick">
        <person layout="standard" name="Pierre Brenkle"
email="pbrenkle@mycompany.com" phone="4155550015" mobilePhone="4155550016"
fax="4155550017" instantMessager="pbrenkle" position="Architect" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="vacation">
```

```
          <person layout="bothHanging" name="Marianne Beach"
email="mbeach@mycompany.com" phone="4155550018" mobilePhone="4155550019"
fax="4155550020" instantMessager="mbeach" position="Developer" location="San
Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
            <person layout="standard" name="Anne Caretta"
email="acaretta@mycompany.com" phone="4155550021" mobilePhone="4155550022"
fax="4155550023" instantMessager="acaretta" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="unknown"/>

            <person layout="standard" name="Jacob Boute"
email="jboute@mycompany.com" phone="4155550024" mobilePhone="4155550025"
fax="4155550026" instantMessager="jboute" position="Developer" location="San
Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>
            <person layout="standard" name="Robert Cassard"
email="rcassard@mycompany.com" phone="4155550027" mobilePhone="4155550028"
fax="4155550029" instantMessager="rcassard" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

            <person layout="standard" name="Amelia Barcelo"
email="abarcelo@mycompany.com" phone="4155550030" mobilePhone="4155550031"
fax="4155550032" instantMessager="abarcelo" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

            <person layout="standard" name="Martin Baader"
email="mbaader@mycompany.com" phone="4155550033" mobilePhone="4155550034"
fax="4155550035" instantMessager="mbaader" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="travel">
              <person layout="standard" name="Sarah Dumull"
email="sdumull@mycompany.com" phone="4155550081" mobilePhone="4155550082"
fax="4155550083" instantMessager="sdumull" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
                <person layout="standard" name="Eugenie Baspie"
email="ebaspie@mycompany.com" phone="4155550084" mobilePhone="4155550085"
fax="4155550086" instantMessager="ebaspie" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

              </person>

            </person>
          </person>
          <person layout="rightHanging" name="Francis Avila"
email="favila@mycompany.com" phone="4155550036" mobilePhone="4155550037"
fax="4155550038" instantMessager="favila" position="Developer" location="San
Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
                        <person layout="standard" name="Vitta Dowling"
email="vdowling@mycompany.com" phone="4155550072" mobilePhone="4155550073"
fax="4155550074" instantMessager="vdowling" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

            <person layout="standard" name="John Fleury"
email="jfleury@mycompany.com" phone="4155550075" mobilePhone="4155550076"
fax="4155550077" instantMessager="jfleury" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="travel"/>
          <person layout="standard" name="Lina Bouny" email="lbouny@mycompany.
```

```
com" phone="4155550078" mobilePhone="4155550079" fax="4155550080"
instantMessager="lbouny" position="Developer" location="San Francisco"
businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

            <person layout="standard" name="Vitta Bonche"
email="vbonche@mycompany.com" phone="4155550039" mobilePhone="4155550040"
fax="4155550041" instantMessager="vbonche" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="travel"/>
          </person>
        </person>
        <person layout="standard" name="Bartholomew Dutrey"
email="bdutrey@mycompany.com" phone="4155550042" mobilePhone="4155550043"
fax="4155550044" instantMessager="bdutrey" position="Architect" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
          <person layout="rightHanging" name="Basile Bronpard"
email="bbronpard@mycompany.com" phone="4155550045" mobilePhone="4155550046"
fax="4155550047" instantMessager="bbronpard" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
           <person layout="standard" name="Michel Cabs" email="mcabs@mycompany.
com" phone="4155550048" mobilePhone="4155550049" fax="4155550050"
instantMessager="mcabs" position="Developer" location="San Francisco"
businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>
            <person layout="standard" name="Elicia Christmann"
email="echristmann@mycompany.com" phone="4155550051" mobilePhone="4155550052"
 fax="4155550053" instantMessager="echristmann" position="Developer"
location="San Francisco" businessUnit="rnd" timeZone="GMT-8"
presenceIndicator="present"/>
            <person layout="standard" name="Patrick Brill"
email="pbrill@mycompany.com" phone="4155550054" mobilePhone="4155550055"
fax="4155550056" instantMessager="pbrill" position="Developer" location="San
Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>
            <person layout="standard" name="Philo Canning"
email="pcanning@mycompany.com" phone="4155550057" mobilePhone="4155550058"
fax="4155550059" instantMessager="pcanning" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

            <person layout="standard" name="Allan Smith Arnault"
email="aarnault@mycompany.com" phone="4155550060" mobilePhone="4155550061"
fax="4155550062" instantMessager="aarnault" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present"/>

          </person>
          <person layout="rightHanging" name="Rene Bichler"
email="rbichler@mycompany.com" phone="4155550063" mobilePhone="4155550064"
fax="4155550065" instantMessager="rbichler" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="travel">
            <person layout="rightHanging" name="Sarah Dumull"
email="sdumull@mycompany.com" phone="4155550081" mobilePhone="4155550082"
fax="4155550083" instantMessager="sdumull" position="Developer" location="San
 Francisco" businessUnit="rnd" timeZone="GMT-8" presenceIndicator="present">
            </person>
          </person>
        </person>
      </person>;
```

```
    ]]>

  </mx:Script>
    <ilog:OrgChart id="chart" width="100%" height="100%" dataProvider="{xml}"

      initialPosition="fitToContents"/>
   <mx:Button click="doPrint()" label="Print"/>
</mx:Application>
```

The following code shows the custom print component file for the multipage printing example.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex"
  visible="false" includeInLayout="false" horizontalAlign="center"
  creationComplete="showLabel(leftLabel); showLabel(topLabel)">
  <mx:Script>
    <![CDATA[
      [Bindable]
      public var row:Number = 1;
      [Bindable]
      public var column:Number = 1;

      public function showLabel(label:Label, value:Boolean = false):void
      {
        label.visible = label.includeInLayout = false;
        validateNow();
      }
    ]]>
  </mx:Script>

  <mx:Label text="TOP" id="topLabel"/>
  <mx:HBox width="100%" height="100%" verticalAlign="middle">
    <mx:Label text="LEFT" id="leftLabel"/>
    <mx:Canvas width="100%" height="100%">
      <ilog:PrintOrgChart width="100%" height="100%" id="chart"
        printingScalingFactor="1.0" allowBoxCut="false" scale="2">
            <ilog:itemRenderer>
              <mx:Component>
                <ilog:OrgChartItemRenderer highlightDuration="500"
fontAntiAliasType="advanced" fontGridFitType="none"
                  borderColor="0"/>
              </mx:Component>
            </ilog:itemRenderer>
      </ilog:PrintOrgChart>
      <mx:Label color="0xCCCCCC" text="{'Row: ' + row + ' Column: ' + column}
"/>
    </mx:Canvas>
  </mx:HBox>
</mx:VBox>
```

Radar Charts

Describes the use of IBM ILOG Elixir radar charts with Adobe® Flex® 3.

In this section

Introduction to radar charts
Describes the uses and layout of radar charts with an example.

Radar chart architecture
Describes the architecture of the radar chart classes.

Creating a radar chart control
Describes a radar chart control and gives an example of the code.

Configuring radar chart data
Describes how to configure radar chart data.

Radar chart axes
Describes the axes of radar charts.

Series in radar charts
Describes the series types that can be used with radar charts.

Radar chart axis renderers
Describes radar chart axis renderers and gives an example of the code.

Radar chart data renderers
Describes the standard Adobe® Flex® Builder™ Professional radar chart data renderers and two specific radar chart data renderers; explains series type/renderer compatibility.

Managing radar chart events
Describes the management of events on radar charts.

User interaction with radar charts
Describes the ways that you can interact with a radar chart.

Using effects in a radar chart
Describes the use of radar chart effects.

Introduction to radar charts

Radar charts are useful when you want to compare a limited set of unrelated factors such as speed, height, or weight. A radar chart has two axes: a radial axis and an angular axis that handles categories. In a radar chart, a point close to the center on any axis indicates a low value and a point near the perimeter indicates a high value.

You can customize the look and feel of a radar chart, including the two radial axis renderers, the angular axis renderer, and the radial and angular grids. A radar chart can contain series of the following types:

- Line
- Plot
- Bubble
- Column

The different series types can be combined into one radar chart.

The `type` property controls the way the chart is renderered. This property is used by `RadarGridLines` and `AngularAxisRenderer`.

The value of `type` can be:

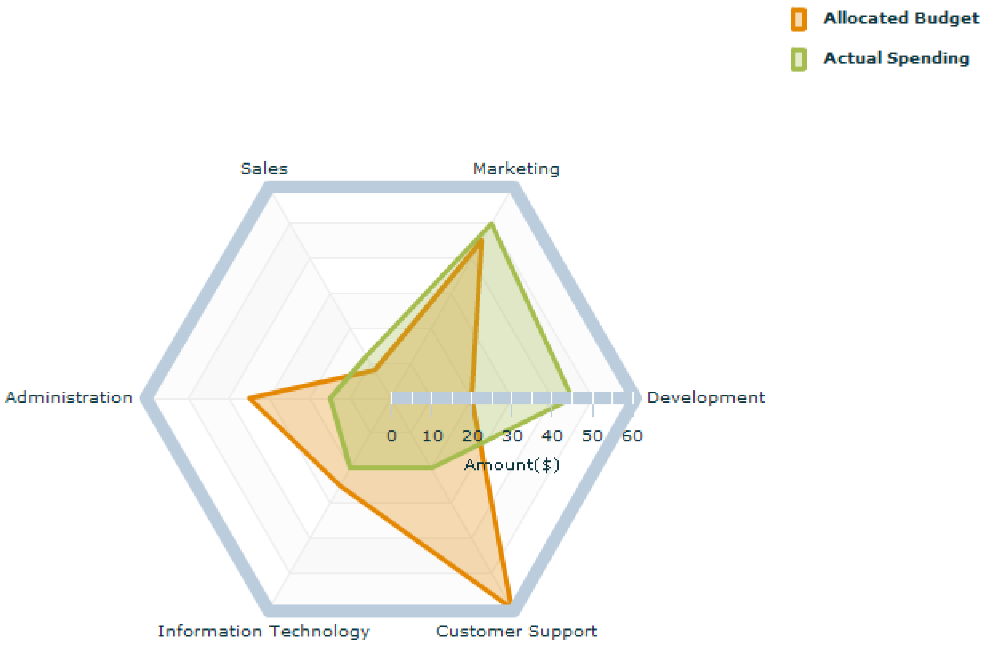
Circular
The grid and the angular axis renderer are made up of concentric circles.

Polygonal
The grid and the angular axis renderer are made up of concentric polygons.

IBM ILOG Elixir radar charts are based on Adobe® Flex® Builder™ Professional and to be fully functional the following features require its installation:

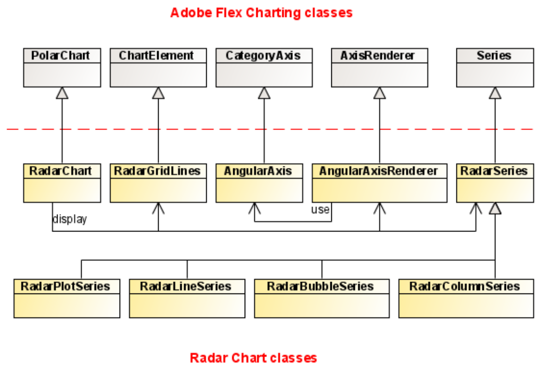
- Animation
- Legend
- Alternate axes (`LogAxis`, `DateTimeAxis`, `CategoryAxis`)
- Polar data canvas
- Alternate item renderers (`TriangleItemRender`, `CircleItemRenderer`...)

The following example of a radar chart shows two line data series representing the allocated budget and the actual spending for a number of different departments in a company. The radial axis represents the cost and the angular axis the departments of the company.


Allocated Budget
Actual Spending
Sales
Marketing
Administration
Development
0
10
20
30
40
50
60
Amount($)
Information Technology
Customer Support

Radar chart architecture

The IBM ILOG Elixir radar chart classes rely on the Adobe® Flex® Builder™ Professional base classes as shown in the following Unified Modeling Language (UML) figure. This dependency allows you to reuse the same concepts, and in particular the Adobe Flex Builder Professional axes, legends, effects, and data canvas, which can be used as is.



The radar chart classes shown in the UML diagram are described in the following table.

Main radar chart classes

Radar chart class	Description
`RadarChart`	The top level class for radar charts. This class inherits directly from the `PolarChart` class. It is the top-level object to be used in MXML.
`RadarGridLines`	Corresponds to the `GridLines` class in Adobe Flex Builder Professional Cartesian charts. This class draws lines radially and concentrically, as for `GridLines`. The `RadarGridLines` class has the following settable properties: `radialFill`, `radialAlternateFill`, `angularFill`, `angularAlternateFill`, `radialStroke`, and `angularStroke`.
`AngularAxis`	The perimeter axis of the chart. This is graphically represented by the `AngularAxisRenderer` class. The `AngularAxis` class inherits from the standard `CategoryAxis`.
`AngularAxisRenderer`	Draws the angular axis. Most of the style properties are attached to this class, for example, the `stroke`, `tickLength`,

Radar chart class	Description
	`tickPlacement`, and `canDropLabel` properties.
`RadarSeries`	The base class for radar series. You do not use this class directly, but use instead the subclasses `RadarPlotSeries`, `RadarLineSeries`, `RadarBubbleSeries`, and `RadarColumnSeries`.

Creating a radar chart control

In IBM ILOG Elixir a radar chart control follows the architecture and paradigms of the Adobe® Flex® Builder™ Professional controls as closely as possible. You can use a radar chart like any other Adobe Flex Builder Professional type, for example, `LineChart`, `ColumnChart` or `PlotChart`.

An example of a radar chart control is given below.

```
<ilog:RadarChart dataProvider="{temperature}">
   <ilog:angularAxis>
     <ilog:AngularAxis categoryField="Month" displayName="Month"/>
   </ilog:angularAxis>
   <ilog:radialAxis>
     <mx:LinearAxis  displayName="Temperature (˚F)"/>
   </ilog:radialAxis>
   <ilog:series>
     <series:RadarLineSeries dataField="London" displayName="London"/>
     <series:RadarLineSeries dataField="Sydney" displayName="Sydney"/>
   </ilog:series>
</ilog:RadarChart>
```

Configuring radar chart data

The chart data is defined by the fields specified in the data series of the chart.

For more information about radar chart data, see *Defining chart data* in *Getting Started with Flex 3*.

For more information on data provider types, see *Types of chart data* in the *Flex 3 Developer's Guide*.

To use the data from a data provider in your radar chart control, you must map the `categoryField` and `dataField` properties of the chart series to fields in the data provider. The `categoryField` property defines the data for the angular axis, and the `dataField` property defines the data for the radial axis.

If the angular axis already specifies the `categoryField` of the series, you do not need to specify it again so you specify only the `dataField` property.

For example, assuming your data provider has the following structure:

```
{Month: "Feb", Profit: 1000, Expenses: 200}
```

The angular axis is defined as follows:

```
<ilog:AngularAxis categoryField="Month"/>
```

You can use the `Profit` or `Expenses` field by mapping the `dataField` property of the series object to one of these fields as follows:

```
<ilog:RadarLineSeries dataField="Profit"/>
<ilog:RadarLineSeries dataField="Expenses"/>
```

Radar chart axes

A radar chart has two axes: angular and radial:

Angular axis

The angular axis corresponds to the perimeter of the chart and takes values that are categories rather than numerics. This axis maps a set of values such as stock ticker symbols, state names, or demographic categories to equal angles around the center of the chart. The `angularAxis` property of the `RadarChart` class must be a subclass of the `CategoryAxis` class. By default it is set to an instance of the `AngularAxis` class.

Radial axis

The radial axis corresponds to the radius of the chart. This axis maps numeric data to a distance between the center and the perimeter of the chart. The `radialAxis` property of the `RadarChart` class must be a subclass of the `NumericAxis` class. By default it is set to an instance of the `LinearAxis` class. You can use the following default Adobe® Flex® Builder™ Professional classes that inherits from `NumericAxis`:

- `DateTimeAxis`: This class maps time-based values, such as hours, days, weeks, or years, along a chart axis.
- `LogAxis`: This class maps numeric data to the axis logarithmically.

By default, a series uses the default axes of a chart, but you can change the axes of a particular series by using the `angularAxis` and `radialAxis` properties of the `RadarSeries` class (the base class for all radar series). Typically, you can change the `radialAxis` of a series to allow comparison of the series with a different range of data.

For more information on chart axes, see the *Flex 3 Developer's Guide*.

Series in radar charts

Describes the series types that can be used with radar charts.

In this section

Radar line series
Describes the use of a radar line series in a radar chart.

Radar bubble series
Describes the use of a radar bubble series in a radar chart.

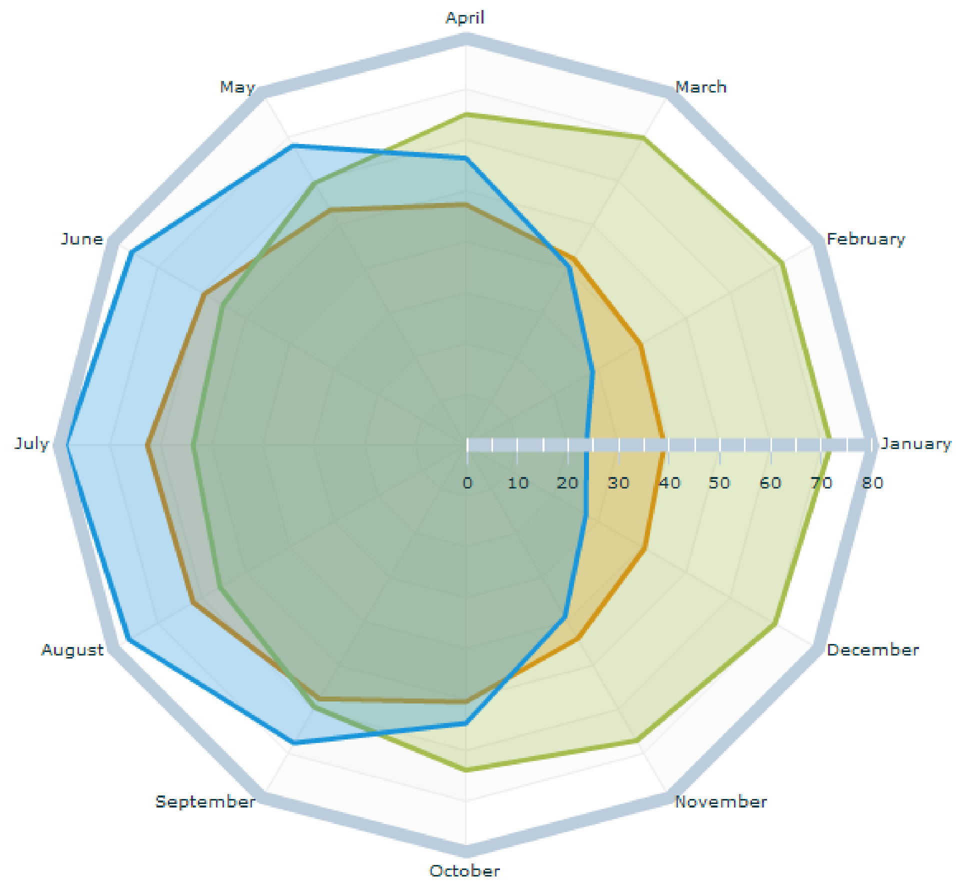
Radar plot series
Describes the use of a radar plot series in a radar chart.

Radar column series
Describes the use of a radar column series in a radar chart.

Radar line series

The `RadarLineSeries` class represents data as a series of points connected by a continuous line that draws a polygon around the center of the chart. You can use an icon or symbol to represent each data point or show a simple line without icons. You can choose to fill the polygon by using the `areaFill` style property.

The following figure shows an example of a radar chart with a line series type. The chart shows the average temperatures for London, Sydney, and Beijing on the radial axis, and the months of the year on the angular axis.



The following code displays the radar line series chart shown in the figure:

```
<?xml version="1.0" ?>
<!--  Simple example to demonstrate the RadarChart control with
 RadarLineSeries.
 -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
```

```
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
<![CDATA[
import mx.collections.ArrayCollection;
[Bindable]
   public var temperature:ArrayCollection = new ArrayCollection([
        {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
        {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
        {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
        {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
        {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
        {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
        {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
        {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
        {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
        {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
        {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
        {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}
]);


  ]]>
  </mx:Script>
<mx:Panel width="100%" height="100%" title="Radar Line Series Example"
layout="horizontal">
  <ilog:RadarChart id="radar" width="100%" height="100%"
   dataProvider="{temperature}" showDataTips="true">
  <ilog:angularAxis>
   <ilog:AngularAxis categoryField="Month" displayName="Month" />
  </ilog:angularAxis>
  <ilog:radialAxis>
   <mx:LinearAxis baseAtZero="true" displayName="Temperature (Ã,Â°F)" />
  </ilog:radialAxis>
  <ilog:series>
    <ilog:RadarLineSeries dataField="London" displayName="London" />
    <ilog:RadarLineSeries dataField="Sydney" displayName="Sydney" />
    <ilog:RadarLineSeries dataField="Beijing" displayName="Beijing" />
  </ilog:series>
  </ilog:RadarChart>
  <mx:Legend dataProvider="{radar}" />
</mx:Panel>
</mx:Application>
```

Radar bubble series

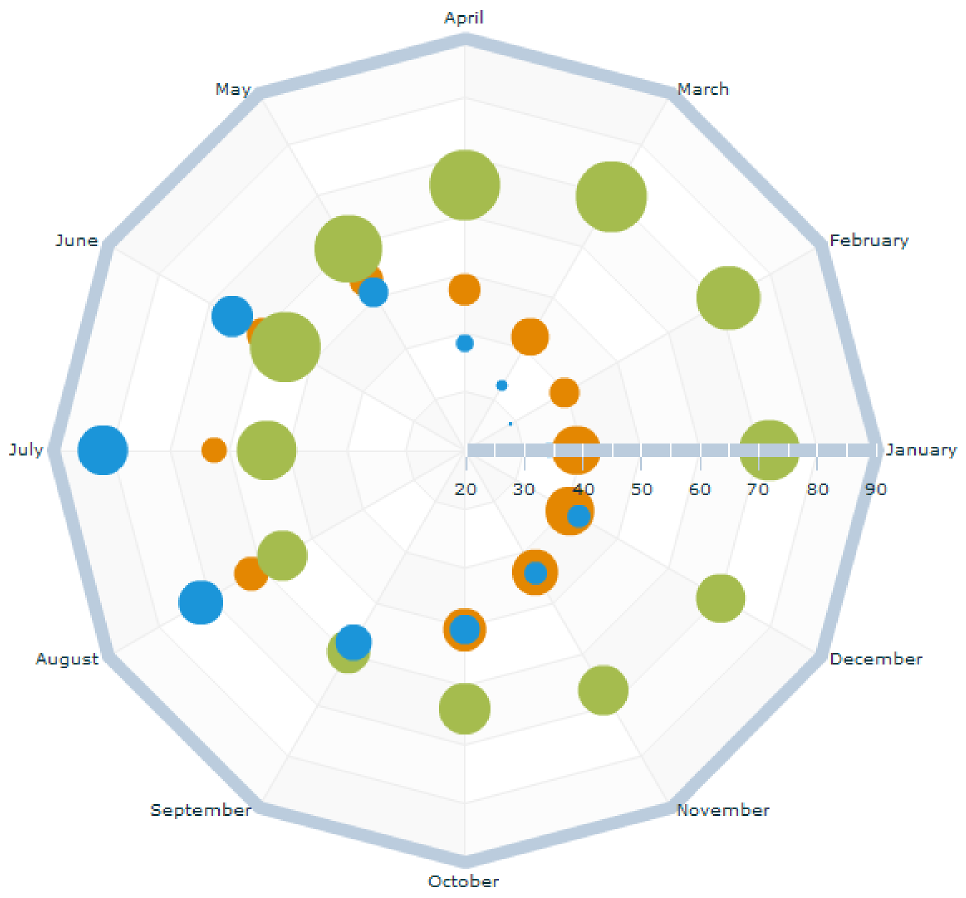
You can use the `RadarBubbleSeries` class to represent data with three values for each data point:

- A category that determines its position along the angular axis.
- A value that determines its position along the radial axis.
- A value that determines the size of the chart symbol relative to the other data points on the chart.

The `radiusField` specifies the field of the data provider that determines the radius of each symbol relative to the other data points in the chart. This property is *mandatory*.

The `<ilog:RadarBubbleSeries>` tag takes an additional attribute, `maxRadius`. This property specifies the maximum radius of the largest chart item, in pixels, as for Cartesian bubble charts. The data point with the largest value is assigned this radius and all other data points are assigned a smaller radius based on their value relative to the largest value. The default value is 30 pixels.

The following figure shows an example of a radar chart with a bubble series type. The chart shows the average temperatures for London, Sydney, and Moscow on the radial axis, and the months of the year on the angular axis. The corresponding rainfall is also displayed on the radial axis and is determined by the size of the bubble.



The following code displays the radar bubble series chart shown in the figure.

```
<?xml version="1.0" ?>
 <!--  Simple example to demonstrate the RadarChart control with
  RadarBubbleSeries.
  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
<![CDATA[
import mx.collections.ArrayCollection;
[Bindable]
  public var london:ArrayCollection = new ArrayCollection([
        {Month:"January", Temperature:39, Rainfall:3.1},
        {Month:"February", Temperature:39.6, Rainfall:2.0},
        {Month:"March", Temperature:42.3, Rainfall:2.4},
        {Month:"April", Temperature:47.3, Rainfall:2.1},
        {Month:"May", Temperature:53.4, Rainfall:2.2},
        {Month:"June", Temperature:59.4, Rainfall:2.2},
        {Month:"July", Temperature:62.6, Rainfall:1.8},
        {Month:"August", Temperature:61.9, Rainfall:2.2},
        {Month:"September", Temperature:57.6, Rainfall:2.2},
        {Month:"October", Temperature:50.5, Rainfall:2.7},
        {Month:"November", Temperature:43.9, Rainfall:2.9},
        {Month:"December", Temperature:40.6, Rainfall:3.1}
     ]);

  [Bindable]
  public var sydney:ArrayCollection = new ArrayCollection([
        {Month:"January", Temperature:71.8, Rainfall:4.1},
        {Month:"February", Temperature:71.8, Rainfall:4.4},
        {Month:"March", Temperature:69.8, Rainfall:5.2},
        {Month:"April", Temperature:65.1, Rainfall:5.1},
        {Month:"May", Temperature:59.5, Rainfall:4.8},
        {Month:"June", Temperature:55.2, Rainfall:5.1},
        {Month:"July", Temperature:53.6, Rainfall:4.0},
        {Month:"August", Temperature:55.8, Rainfall:3.2},
        {Month:"September", Temperature:59.5, Rainfall:2.7},
        {Month:"October", Temperature:63.9, Rainfall:3.3},
        {Month:"November", Temperature:67.1, Rainfall:3.2},
        {Month:"December", Temperature:70.2, Rainfall:3.1}
     ]);

  [Bindable]
  public var moscow:ArrayCollection = new ArrayCollection([
        {Month:"January", Temperature:34.4, Rainfall:1.4},
        {Month:"February", Temperature:29.0, Rainfall:1.1},
        {Month:"March", Temperature:32.7, Rainfall:1.3},
        {Month:"April", Temperature:38.2, Rainfall:1.5},
        {Month:"May", Temperature:51, Rainfall:2.0},
        {Month:"June", Temperature:65.6, Rainfall:2.6},
        {Month:"July", Temperature:81.5, Rainfall:3.2},
        {Month:"August", Temperature:71.8, Rainfall:2.8},
        {Month:"September", Temperature:57.7, Rainfall:2.3},
        {Month:"October", Temperature:50.4, Rainfall:2.0},
        {Month:"November", Temperature:44.1, Rainfall:1.7},
        {Month:"December", Temperature:42.4, Rainfall:1.7}
     ]);


  ]]>
  </mx:Script>
<mx:Panel width="100%" height="100%" title="Radar Bubble Series Example"
layout="horizontal">
  <ilog:RadarChart id="radar" width="100%" height="100%" showDataTips="true">

  <ilog:angularAxis>
    <ilog:AngularAxis dataProvider="{london}" categoryField="Month"
     displayName="Month" />
  </ilog:angularAxis>
  <ilog:radialAxis>
    <mx:LinearAxis minimum="20" maximum="90" displayName="Temperature (Â°F)"
```

```
/>
  </ilog:radialAxis>
  <ilog:series>
  <ilog:RadarBubbleSeries maxRadius="30" dataProvider="{london}"
   dataField="Temperature" radiusField="Rainfall" displayName="London">
  <ilog:radiusAxis>
    <mx:LogAxis id="rainfallAxis" displayName="Rainfall (inches)" />
  </ilog:radiusAxis>
  </ilog:RadarBubbleSeries>
  <ilog:RadarBubbleSeries maxRadius="30" dataProvider="{sydney}"
   dataField="Temperature" radiusField="Rainfall" displayName="Sydney"
    radiusAxis="{rainfallAxis}" />
  <ilog:RadarBubbleSeries maxRadius="30" dataProvider="{moscow}"
   dataField="Temperature" radiusField="Rainfall" displayName="Moscow"
    radiusAxis="{rainfallAxis}" />
  </ilog:series>
  </ilog:RadarChart>
  <mx:Legend dataProvider="{radar}" />
  </mx:Panel>
</mx:Application>
```

Radar plot series

You can use the `RadarPlotSeries` class to represent data where each data point has one category that determines its position along the angular axis, and one value that determines its position along the radial axis. Using the data series renderers, you can define the shape that IBM ILOG Elixir displays at each data point.

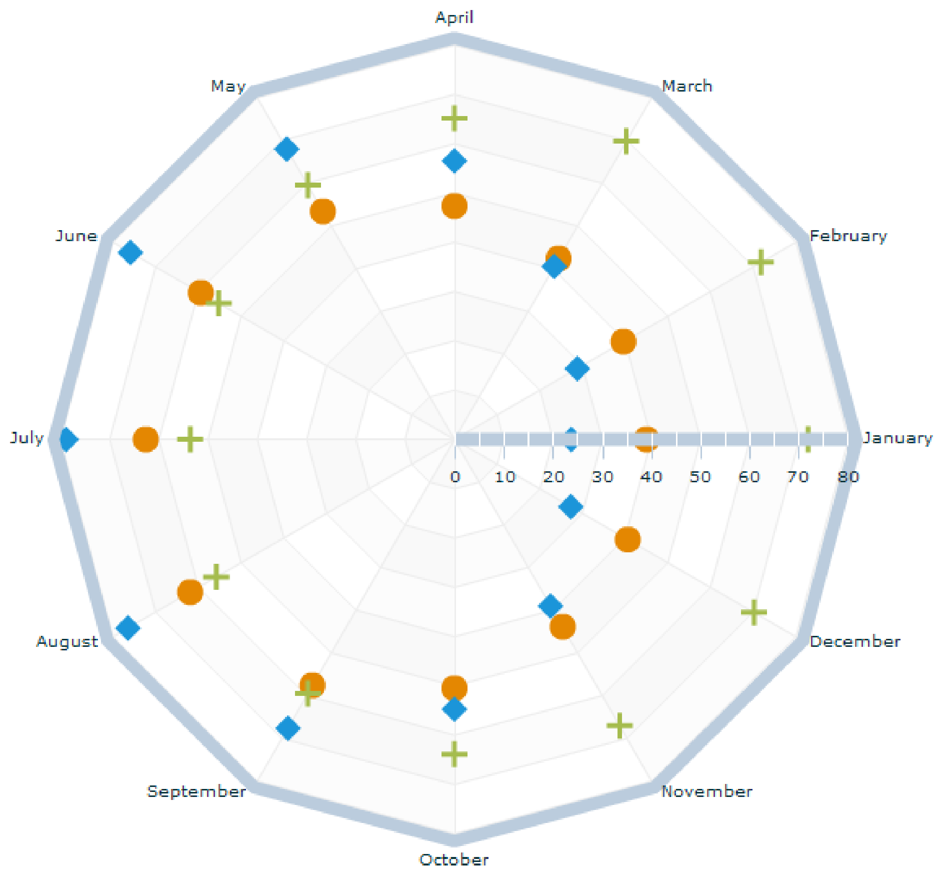
By default, IBM ILOG Elixir displays the first data series in the chart as a diamond at each point. When you define multiple data series in a chart, IBM ILOG Elixir rotates the shape for the series starting with a diamond, then a circle, and then a square. If you have more series than default renderers, IBM ILOG Elixir begins again with the diamond.

The diamond shape, like the other shapes, is defined by a renderer class. The renderer classes that define these shapes are located in the Adobe® Flex® `mx.charts.renderers` package. The circle is defined by the `CircleItemRenderer` class. To use a renderer other than the default, you can set the `itemRenderer` style property as shown in the following code:

```
<ilog:RadarPlotSeries dataField="London"
   itemRenderer="mx.charts.renderers.DiamondItemRenderer" />
```

For more information about renderers, see *Radar chart data renderers*.

The following figure shows an example of a radar chart with a plot series type. The chart shows the average temperatures for London, Sydney, and Beijing on the radial axis, and the months of the year on the angular axis.



The following code displays the radar plot series chart shown in the figure.

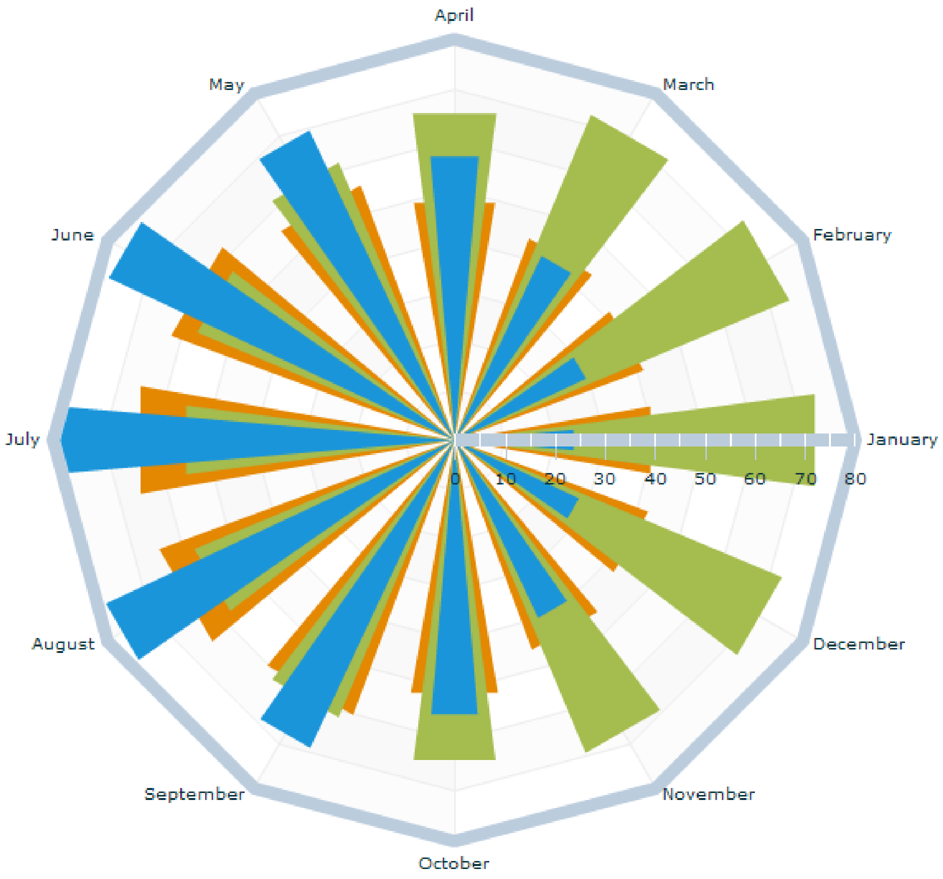
```
<?xml version="1.0" ?>
<!--  Simple example to demonstrate the RadarChart control with
 RadarPlotSeries.
 -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
<![CDATA[
import mx.charts.renderers.CrossItemRenderer;
import mx.charts.renderers.CircleItemRenderer;
import mx.collections.ArrayCollection;
  [Bindable]
  public var temperature:ArrayCollection = new ArrayCollection([
        {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
        {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
        {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
        {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
        {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
        {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
        {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
        {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
        {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
        {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
        {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
        {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}
     ]);


  ]]>
  </mx:Script>
<mx:Panel width="100%" height="100%" title="Radar Plot Series Example"
 layout="horizontal">
  <ilog:RadarChart id="radar" width="100%" height="100%"
   dataProvider="{temperature}" showDataTips="true">
  <ilog:angularAxis>
    <ilog:AngularAxis categoryField="Month" displayName="Month" />
  </ilog:angularAxis>
  <ilog:radialAxis>
    <mx:LinearAxis baseAtZero="true" displayName="Temperature (Â°F)" />
  </ilog:radialAxis>
  <ilog:series>
    <ilog:RadarPlotSeries radius="8" dataField="London" displayName="London"
     itemRenderer="mx.charts.renderers.CircleItemRenderer" />
    <ilog:RadarPlotSeries radius="8" dataField="Sydney" displayName="Sydney"
     itemRenderer="mx.charts.renderers.CrossItemRenderer" />
   <ilog:RadarPlotSeries radius="8" dataField="Beijing" displayName="Beijing"

     itemRenderer="mx.charts.renderers.DiamondItemRenderer" />
  </ilog:series>
  </ilog:RadarChart>
  <mx:Legend dataProvider="{radar}" />
  </mx:Panel>
</mx:Application>
```

Radar column series

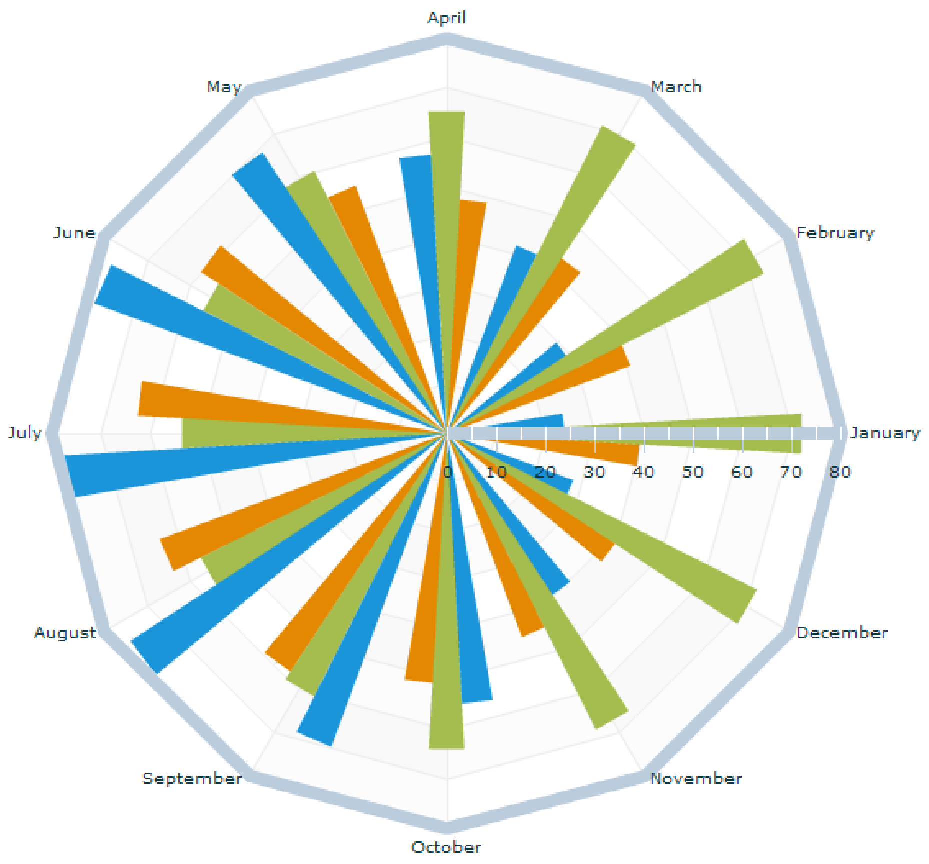
The RadarColumnSeries class represents data as a series of columns whose height is determined by values in the data. In a radar chart, a column is a triangle which starts at the center. By default, if there is more than one column series in the radar chart, the columns are clustered on each category angle of the angular axis. However you can also make a group of columns using the columnGroup property. All column series with the same columnGroup are overlaid. The following code provides an example of how to do this.

```
<ilog:series>
   <ilog:RadarColumnSeries columnGroup="g1" columnWidthRatio="1.0"
      dataField="London" displayName="London"/>
   <ilog:RadarColumnSeries columnGroup="g1" columnWidthRatio="0.75"
      dataField="Sydney" displayName="Sydney"/>
   <ilog:RadarColumnSeries columnGroup="g1" columnWidthRatio="0.5"
      dataField="Beijing" displayName="Beijing"/>
</ilog:series>
```

The following figure shows an example of a grouped radar column chart. The chart shows the average temperatures for London, Sydney, and Beijing on the radial axis, and the months of the year on the angular axis.



The following figure shows the same example in the form of a clustered radar column chart.



The following code is an example of a clustered radar column series chart that displays the chart shown in the figure.

```
<?xml version="1.0" ?>
<!--  Simple example to demonstrate the RadarChart control with
 RadarColumnSeries.
 -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
<mx:Script>
<![CDATA[
import mx.collections.ArrayCollection;
   [Bindable]
   public var temperature:ArrayCollection = new ArrayCollection([
        {Month:"January", London:39, Sydney:71.8, Beijing:23.7},
        {Month:"February", London:39.6, Sydney:71.8, Beijing:28.8},
        {Month:"March", London:42.3, Sydney:69.8, Beijing:40.5},
        {Month:"April", London:47.3, Sydney:65.1, Beijing:56.5},
        {Month:"May", London:53.4, Sydney:59.5, Beijing:68},
        {Month:"June", London:59.4, Sydney:55.2, Beijing:75.9},
        {Month:"July", London:62.6, Sydney:53.6, Beijing:78.8},
        {Month:"August", London:61.9, Sydney:55.8, Beijing:76.5},
        {Month:"September", London:57.6, Sydney:59.5, Beijing:67.6},
        {Month:"October", London:50.5, Sydney:63.9, Beijing:54.7},
        {Month:"November", London:43.9, Sydney:67.1, Beijing:39},
        {Month:"December", London:40.6, Sydney:70.2, Beijing:27.3}
     ]);


  ]]>
  </mx:Script>
<mx:Panel width="100%" height="100%" title="Radar Column Series Example"
 layout="horizontal">
  <ilog:RadarChart id="radar" width="100%" dataProvider="{temperature}"
   height="100%" showDataTips="true">
  <ilog:angularAxis>
    <ilog:AngularAxis categoryField="Month" displayName="Month" />
  </ilog:angularAxis>
  <ilog:radialAxis>
    <mx:LinearAxis displayName="Temperature (Â°F)" />
  </ilog:radialAxis>
  <ilog:series>
    <ilog:RadarColumnSeries dataField="London" displayName="London" />
    <ilog:RadarColumnSeries dataField="Sydney" displayName="Sydney" />
    <ilog:RadarColumnSeries dataField="Beijing" displayName="Beijing" />
  </ilog:series>
  </ilog:RadarChart>
  <mx:Legend dataProvider="{radar}" />
</mx:Panel>
</mx:Application>
```

Radar chart axis renderers

A radar chart allows you to use one angular axis renderer (typically an `AngularAxisRenderer` instance) and a maximum of two radial axis renderers (one from the center to the top and one from the center to the right).

A radar chart has two arrays of axis renderers: `radialAxisRenderers` and `angularAxisRenderers`.

The following code shows how to set two radial axis renderers that are associated with the same axis instance:

```
<ilog:RadarChart>
     <ilog:radialAxisRenderers>
       <mx:AxisRenderer horizontal="true"/>
       <mx:AxisRenderer horizontal="false"/>
     </ilog:radialAxisRenderers>
 </ilog:RadarChart>
```

Radar chart data renderers

Standard chart renderers

The standard Adobe® Flex® Builder™ Professional renderers are:

- ♦ `CircleItemRenderer`
- ♦ `CrossItemRenderer`
- ♦ `DiamondItemRenderer`
- ♦ `TriangleItemRenderer`

These renderers are used by radar charts to display an item at a point location. For more details of the standard Adobe Flex Builder Professional renderers, see the *Flex 3 Language Reference*.

Specific radar chart renderers

There are two specific radar chart renderers:

- ♦ `RadarColumnRenderer`: A chart item renderer implementation that fills a triangular area. If no minimum value is defined, the triangle starts at the origin (center) of the radar chart, otherwise the triangle is cut at the point corresponding to the minimum value given.
- ♦ `RadarLineRenderer`: A simple implementation of a line segment renderer that is used by `RadarLineSeries` objects. This renderer is like a vertical cut through a polyline chart, which here corresponds to a polygon around the center. If the `areaFill` style property is set on the corresponding series, the polygon is filled accordingly.

You can use the per-item fill API with standard chart renderers and with `RadarColumnRenderer`. In this way you can dynamically choose the colors for the individual chart items in a chart. This helps you to visually differentiate items and easily summarize the visual message that you need to receive.

Series type/renderer compatibility

All series except `RadarColumnSeries` are able to draw plot renderers. See *Radar plot series* for more information.

The series type/renderer compatibility is shown in the following table:

Series type/renderer compatibility

Radar Series	Renderers	Renderer instanc
`RadarLineSeries`	lineSegmentRenderer	`RadarLineRender` area rendering)
	itemRenderer	CircleItemRenderer, DiamondItemRende
`RadarPlotSeries`	itemRenderer	CircleItemRenderer, DiamondItemRende
`RadarBubbleSeries`	itemRenderer	CircleItemRenderer, DiamondItemRende
`RadarColumnSeries`	itemRenderer	`RadarColumnRend` CrossItemRenderer, TriangleItemRendere

Managing radar chart events

The radar chart control manages user interaction events. The radar chart component exposes all the Adobe® Flex® Builder™ Professional events, but does not extend them. The available events are described in the section *About charting events* in the *Flex 3 Developer's Guide*.

The following code shows how to listen for an `itemClick` event on chart columns that calls an `itemClickAction` function:

```
<ilog:RadarChart height="100%" width="100%"
  itemClick="itemClickAction()"
  dataProvider="{stockDataAC}">
```

User interaction with radar charts

Radar charts support the Adobe® Flex® Builder™ Professional 3.0 interactions. These provide the chart interaction framework that allows you to select chart items and regions. To allow selection on a given radar chart, set the `selectionMode` property to `single` or to `multiple` if multiple items can be selected. The following code shows how to do this:

```
<ilog:RadarChart height="100%" width="100%"
  selectionMode="multiple"
  dataProvider="{stockDataAC}">
```

Selection

You can select as shown in the following table.

Selection actions

Function	Action
Select	Click a point in the radar chart.
Extend a selection	Hold down the CTRL key and click the items.
Reduce a selection	Hold down the CTRL key and click an already selected item to remove it from the selection.
Clear a selection	Click anywhere in the chart background, but not on radar chart data.

Navigation

Keyboard accessibility is also provided for navigation through radar chart items. The keyboard actions for radar charts are shown in the following table.

Keyboard actions

Keyboard	Action
Right/left arrow keys	Navigate through the angular axis categories.
Up/down arrow keys	Navigate through the different series items of the current category.

For more information, see *Handling user interactions with charts* and *Selecting chart items* in the *Flex 3 Developer's Guide*.

Using effects in a radar chart

Radar chart controls support the standard Adobe® Flex® Builder™ Professional series effects: *slide*, *zoom*, and *interpolate*.

The SeriesSlide effect

The `SeriesSlide` effect allows you to make data slide in and out of a screen when the data changes. This creates animation effects using the two sets of values. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesSlide effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesSlide` effect.

```
<mx:SeriesSlide duration="1000" direction="down" elementOffset="0"
   id="slideDown" />
<ilog:RadarLineSeries dataField="Temperature" displayName="Temperature"
   showDataEffect="slideDown" hideDataEffect="slideDown" />
```

The SeriesZoom effect

The `SeriesZoom` effect allows you to implode and explode chart data in and out of the focal point that you specify to create chart animations. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesZoom effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesZoom` effect.

```
<mx:SeriesZoom duration="1000" elementOffset="50" id="zoom" relativeTo="chart"

  />
<ilog:RadarColumnSeries dataField="Temperature" displayName="Temperature"
  showDataEffect="zoom" hideDataEffect="zoom" />
```

The SeriesInterpolate effect

The `SeriesInterpolate` effect allows you to move the graphics that represent the existing data in a series to new points. Instead of clearing the chart and then repopulating it, as with `SeriesZoom` and `SeriesSlide`, this effect keeps the data on the screen at all times. You do not lose the context on data changes. The effect is not set on the chart itself but on the series. For more information, see *Using the SeriesInterpolate effect* in the *Flex 3 Developer's Guide*.

The following example shows how to code the `SeriesInterpolate` effect.

```
<mx:SeriesInterpolate duration="1000" id="interpolate" elementOffset="10" />
<ilog:RadarLineSeries dataField="Temperature"
  displayName="Temperature" showDataEffect="interpolate" />
```

Treemaps

Describes the use of IBM ILOG Elixir treemaps with Adobe® Flex® 3.

In this section

Introduction to treemaps
Describes the uses and features of treemaps with an example.

Creating a treemap control
Describes a treemap control and gives an example of the code.

Configuring treemap data
Describes the facilities for configuring your treemap data.

Styling treemaps
Describes the treemap styling properties and gives some examples of property usage.

Examples of property usage
Gives examples of properties used to style treemaps.

Managing treemap events
Describes the management of events on treemaps.

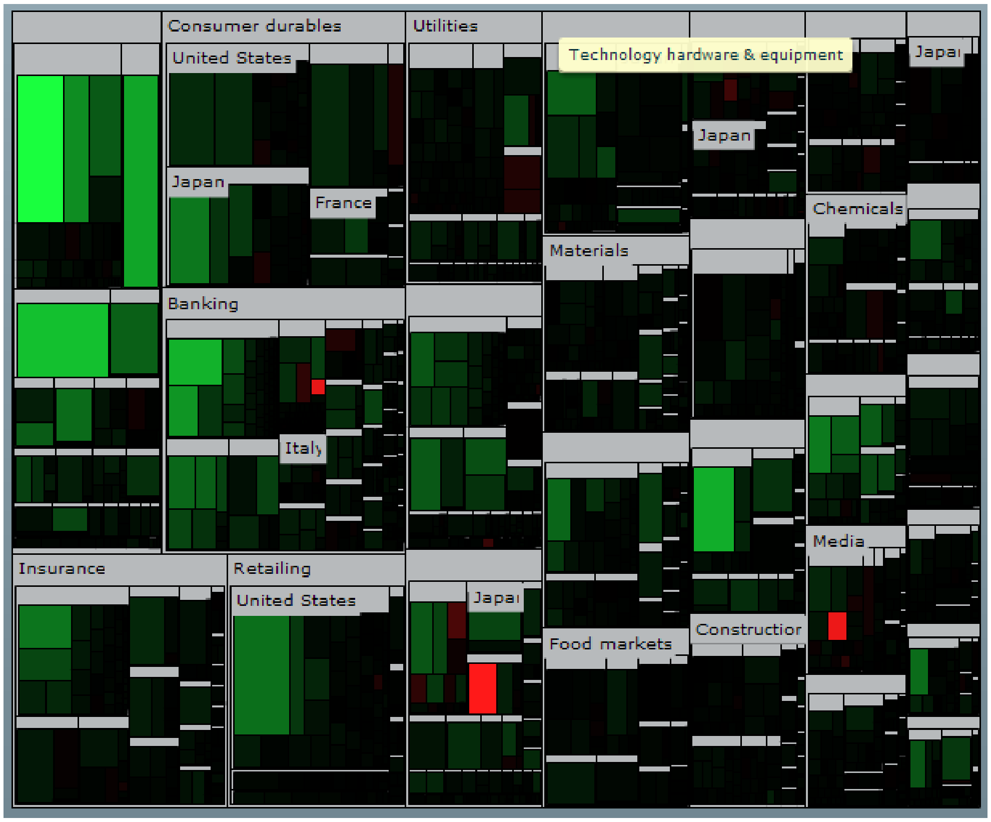
User interaction with treemaps
Describes the ways that you can interact with a treemap.

Treemap legend control
Describes a treemap legend control and gives an example of the code.

Introduction to treemaps

Treemaps are often described as graphical pivot tables that can be used to explore large data sets by using convenient drill-down capabilities. Treemaps reveal data patterns and trends easily. Treemaps rely on data clustering, using areas and color information to represent the data you want to explore.

An example of a treemap is shown below. The treemap shows business sectors at the head of the hierarchy and provides the possibility to drill down to country and then company level.



IBM ILOG Elixir supports `squarified` and `alternating` (also known as slice-and-dice) type algorithms for two-dimensional treemaps, and gives you a great deal of control over the appearance.

In IBM ILOG Elixir treemaps are characterized by the ability to:

- Map the size, color, and label of treemap items to properties in the `dataProvider`.
- Choose which treemap packing algorithm is used to arrange the treemap items.
- Choose either a predefined algorithm for computing the item colors or specify a color using a customizable color function.

- Specify the margins between the treemap items.
- Specify the treemap levels at which labels are to appear.
- Get an event when clicking and hovering over treemap items.
- Navigate within a treemap with visual effects on drill down.

Creating a treemap control

You can define a TreeMap control in MXML using the <ilog:TreeMap> tag. If you intend to refer to this control elsewhere in your MXML, for example, in another tag or in an ActionScript® block, you must specify an id value. A TreeMap control displays tree data in a specific manner; however, it is similar to the Adobe® Flex® Tree control and can be configured in the same way by using a dataProvider as shown in the following code:

```
<ilog:TreeMap width="200" height="400" id="myTreeMap"
              dataProvider="{myDataProvider}" />
```

Configuring treemap data

Describes the facilities for configuring your treemap data.

In this section

Treemap data providers
Describes the possible data providers for treemaps.

Specifying treemap data with field mappings
Describes the field properties for mapping treemap data and how to map the data.

Specifying treemap data with custom functions
Describes the use of custom functions to provide treemap data.

Treemap data providers

A `TreeMap` control gets its data from a hierarchical data provider, such as XML. If the `TreeMap` represents dynamically changing data, you can use an object that implements the `ICollectionView` interface, such as `ArrayCollection` or `XMLListCollection`.

A `TreeMap` control uses a data descriptor to parse and manipulate the contents of the data provider. By default, a `TreeMap` control uses the `DefaultDataDescriptor` class provided by the Adobe® Flex® framework, but you can create your own class and specify it using the `dataDescriptor` property.

The Adobe® Flex® framework `DefaultDataDescriptor` class supports the following types of data:

XML
A string containing valid XML text or any of the following objects containing valid E4X format XML data: `<mx:XML>` or `<mx:XMLList>` compile-time tag, or an XML or `XMLList` object.

IHierarchicalData
An `IHierarchicalData` or `IHierarchicalCollectionView` implementation instance. These interfaces and their implementations are available with Adobe® Flex® Builder(TM) Professional.

Other objects
An array of items or an object that contains an array of items, where the children of a node are contained in an item named *children*.

Collections
An object that implements the `ICollectionView` interface (such as the `ArrayCollection` or `XMLListCollection` classes) and whose data provider conforms to the structure specified in either of these items. The `DefaultDataDescriptor` class includes code that handles collections efficiently. If the data changes dynamically, you must always use a collection as the data provider; otherwise the `TreeMap` control may display obsolete data.

In addition to specifying the `dataProvider` of the `TreeMap` control, you must also map the different fields required to display the control to fields that exist in the `dataProvider` content.

Specifying treemap data with field mappings

The field properties used to display the TreeMap control are described in the following table.

Field properties for the TreeMap control

Property	Default value	Purpose
areaField	area	The value of the field whose name is stored in this property is used to compute the area representing each treemap item.
colorField	color	The value of the field whose name is stored in this property is used in conjunction with the coloring scheme to compute the color of each treemap item.
labelField	label	The value of the field whose name is stored in this property is used to display the label of each treemap item.

The following code contains a single TreeMap control that shows the contents of a mailbox. It shows the field property areaField mapping the area field of the control to the dataProvider attribute total.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http//www.ilog.com/2007/ilog/flex">
   <ilog:TreeMap width="200" height="400"
                 areaField="@total"
                 algorithm="alternating"
                 colorScheme="depth">
      <mx:XMLListCollection id="MailBox">
        <mx:XMLList>
        <folder label="Mail">
          <folder label="Inbox">
            <folder label="Marketing" total="100" read="70"/>
            <folder label="Product Management" total="200" read="99"/>
            <folder label="Personal" total="50" read="50"/>
          </folder>
          <folder label="Outbox">
            <folder label="Professional" total="150" read="100"/>
            <folder label="Personal" total="5" read="100"/>
          </folder>
          <folder label="Spam" total="200" read="1"/>
          <folder label="Sent" total="0" read="100"/>
        </folder>
        </mx:XMLList>
      </mx:XMLListCollection>
   </ilog:TreeMap>
</mx:Application>
```

The tags that represent treemap items in the XML data can have any names. The TreeMap control reads the XML and builds the display hierarchy based on the nested relationship of the items. For information on valid XML structures, hierarchical objects, and data descriptors, including a detailed description of the formats supported by the DefaultDataDescriptor, see *Data descriptors and hierarchical data provider structure* in the *Flex 3 Developer’s Guide*.

Specifying treemap data with custom functions

If the data values for a field in a treemap control are not directly available in one of the fields of the data provider items, you can use the `areaFunction`, `colorFunction`, or `labelFunction` property to specify a custom function to provide the area, color, or label respectively of each treemap item.

For example, you can change the label displayed by the treemap and prefix it with an arrow by using the following code, which specifies the custom function `lFunction` to retrieve the label.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
   <![CDATA[
    public function lFunction(item:Object):String {
      return "->"+item.@label;
    }
   ]]>
  </mx:Script>
  <ilog:TreeMap width="200" height="400"
                 areaField="@total" labelFunction="{lFunction}"
                 labelThreshold="2">
        <!-- data -->
  </ilog:TreeMap>
</mx:Application>
```

Similarly, you can use the `colorFunction` property to specify a custom function that returns a color dependent on the treemap item.

Styling treemaps

Describes the properties that you can set to change the appearance of a `TreeMap` control once it has been created and configured to read and map data.

The algorithm property

The algorithm property affects the arrangement of items.

When you use the `algorithm` property, you can choose either the `squarified` (default value) algorithm that packs items with similar areas together, or the `alternating` algorithm that arranges the items according to their order in the data provider.

The labelThreshold property

The `labelThreshold` property affects the display of labels for treemap items.

The `labelThreshold` property can be used to display labels for treemap items. By default it is set to -1, which means that no labels are drawn. Setting it to 2 means that labels of depth 0, 1, and 2 are displayed.

Labels are only drawn inside each treemap item if there is enough room for them not to overlap on each other or adjacent areas. Label drawing can be customized by using the usual text style properties. The `labelThreshold` property is used in the code given in the section on the margin properties that follows.

The margin properties

The margin properties affect the margins between treemap items.

The size of an item in a treemap is computed automatically from the area information; however, you can use the maximum margin and margin proportion properties to specify which margins appear between items in the right, left, top, and bottom directions. The following example sets the `topMarginProportion` property to 20% of the size of a treemap item. This gives more room at the top of each item so that the label can be displayed inside the margin:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
   <![CDATA[
    public function lFunction(item:Object):String {
      return "->"+item.@label;
    }
   ]]>
  </mx:Script>
  <ilog:TreeMap width="200" height="400"
                 areaField="total" labelFunction="{lFunction}"
                 labelThreshold="2"
                 topMarginProportion="0.2">
        <!-- data -->
  </ilog:TreeMap>
</mx:Application>
```

You can also remove the margin by setting the `maxTopMargin`, `maxBottomMargin`, `maxLeftMargin`, and `maxRightMargin` properties to zero.

The borderColor property

The `borderColor` property affects the border around a treemap item.

The border around a treemap item can be styled using the `borderColor` style property. The following code shows how to do this.

```
<ilog:TreeMap height="100%" width="100%"
  borderColor="0xFF0000"
  dataProvider="{stockDataAC}">
```

The colorScheme property

The `colorScheme` property affects the color computation for a treemap item.

The `colorScheme` style property allows you to choose how the color of each treemap item is computed. There are three types of color scheme:

- The `none` color scheme: in this case the color is taken from the result given by the `colorFunction`, if any.
- The `depth` color scheme: in this case the color is computed from the depth of the item in the `dataProvider` hierarchy and from the `fillColor` style property.
- Other color schemes: in this case the color is computed from the value corresponding to the `colorField` in the `dataProvider` using a different kind of distribution algorithm. Most of the distribution algorithms distribute the color linearly from the minimum to the maximum value of the color attributed to `colorField`. You can change the neutral value of this distribution using the `neutral` property of the treemap.

The default value for `colorScheme` is `sequential`.

The renderFunction property

The renderFunction property affects the rendering of a treemap item.

If the default rendering of an item is a rectangular area filled with the color computed by the `colorScheme` style property and this does not fit in with your needs, you can set a function on the `renderFunction` property that overrides the default drawing. This function takes several parameters that help you to draw the item.

For example:

```
<ilog:TreeMap renderFunction={renderFunction}/>
```

with the following code:

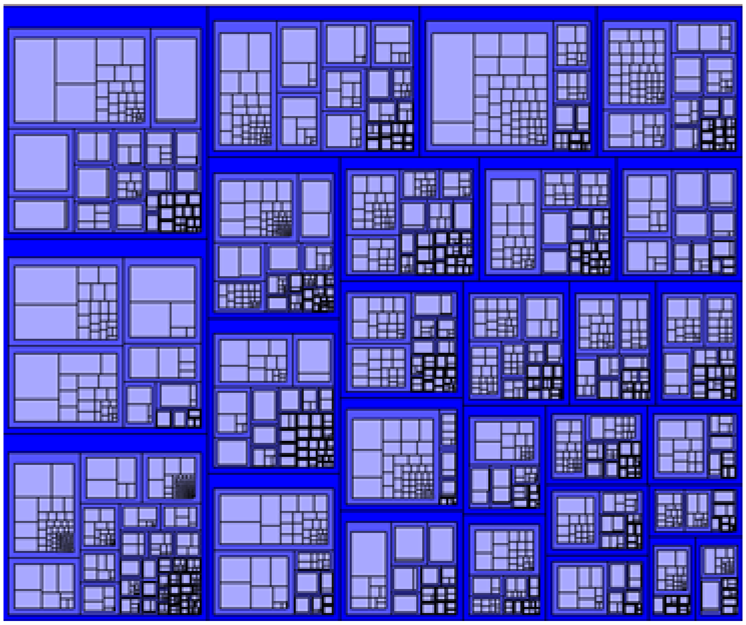
```
function renderFunction(item:Object, graphics:Graphics, rect:Rectangle,
                        fillColor:uint, depth:int, isLeaf:Boolean):void {
      var matr:Matrix = new Matrix();
      matr.createGradientBox(rect.width, rect.height, 0, rect.x, rect.y);
      graphics.beginGradientFill(GradientType.LINEAR,  [fillColor,
         ColorUtil.adjustBrightness(fillColor, 50)],
         [100,100], [0x00, 0xFF], matr, SpreadMethod.PAD);
      graphics.drawRect(rect.x, rect.y, rect.width, rect.height);
      graphics.endFill();
}
```

replaces the default drawing by a gradient based on the color computed by the `colorScheme` style property.

Examples of property usage

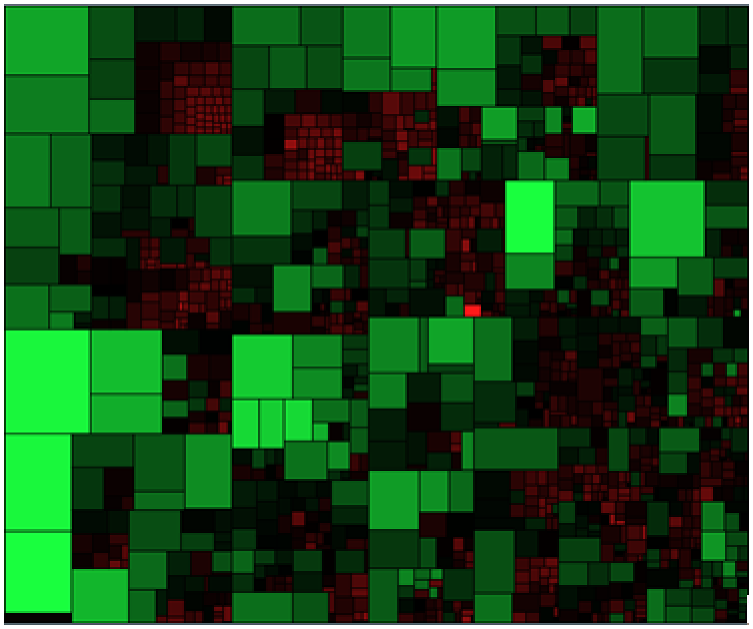
The following figures and code show some examples of treemap rendering displaying the same data set, but with different properties.

Example 1



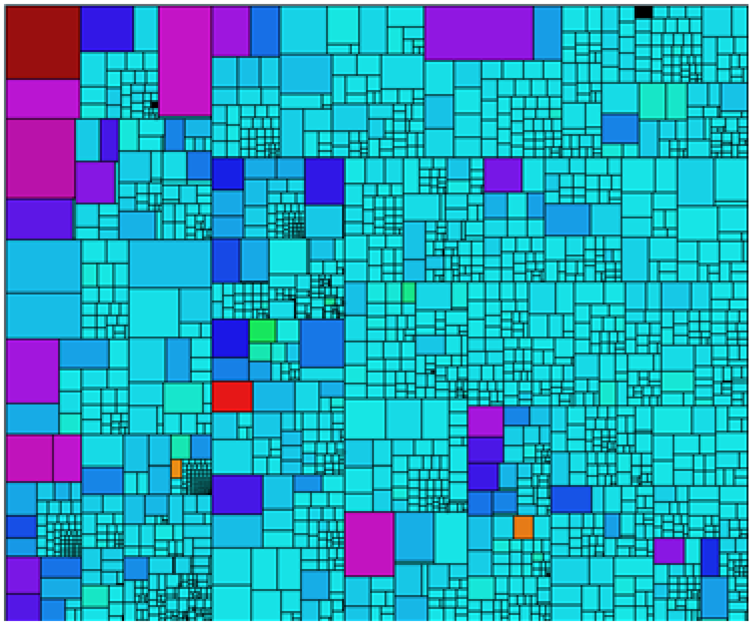
```
<ilog:TreeMap colorScheme="depth"
              fillColor="blue"
              bottomMarginProportion="0.1" topMarginProportion="0.1"
              leftMarginProportion="0.1" rightMarginProportin="0.1"/>
```

Example 2



```
<ilog:TreeMap colorScheme="div-red-green"
              bottomMarginProportion="0" topMarginProportion="0"
              leftMarginProportion="0" rightMarginProportin="0"/>
```

Example 3



```
<ilog:TreeMap colorScheme="qualitative"
              bottomMarginProportion="0" topMarginProportion="0"
              leftMarginProportion="0" rightMarginProportin="0 />
```

Managing treemap events

A TreeMap control can fire several kinds of specific event in response to user interactions. You can create an event handler for each of these events.

For example, if there is a function listening for the itemClick event on the treemap and you click an item, you can access the data for this item.

You can listen for the following treemap data events.

Treemap data events

Treemap data event	Description
change	Indicates that the selection changed as a result of user interaction.
itemClick	Indicates that the user clicked the pointer over a visual item in the control.
itemDoubleClick	Indicates that the user double-clicked the pointer over a visual item in the control.
itemRollOut	Indicates that the user rolled the pointer out of a visual item in the control.
itemRollOver	Indicates that the user rolled the pointer over a visual item in the control.

The treemap events are of type TreeMapEvent.

The following code is an example of how to display a tool tip on each treemap item by listening for the itemRollOver event and the itemRollOut event.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:ilog="http://www.ilog.com/2007/ilog/flex">
  <mx:Script>
   <![CDATA[
   import mx.controls.ToolTip;
   import mx.managers.ToolTipManager;
   import ilog.treemap.TreeMapEvent;

   public var tooltip:ToolTip;

   public function onover(event:TreeMapEvent):void {
     if (event.item) {
       tooltip =
         ToolTipManager.createToolTip(event.item.@label,
           MouseEvent(event.triggerEvent).stageX,
           MouseEvent(event.triggerEvent).stageY)
       as ToolTip;
     }
   }

   public function onout(event:TreeMapEvent):void {
     if (tooltip) {
       ToolTipManager.destroyToolTip(tooltip);
       tooltip = null;
     }
   }
   ]]>
  </mx:Script>
  <ilog:TreeMap width="200" height="400"
                 areaField="total"
                 itemRollOver="onover(event)"
                 itemRollOut="onout(event)"
                 >
        <!-- data -->
  </ilog:TreeMap>
</mx:Application>
```

User interaction with treemaps

When you have set up your treemap application you can execute it. If you click a treemap item, in addition to firing an `itemClick` event, the item is selected and a selection border is drawn around it. You can navigate within the treemap items and customize the treemap selection border color.

For more information, see *Handling user interactions with charts* and *Selecting chart items* in the *Flex 3 Developer's Guide*.

Selection

You can decide whether to select a single treemap item or several items at the same time when you press the CTRL key. This is done by setting the `allowMultipleSelection` property to `true`. The following code shows how to do this.

```
<ilog:Treemap height="100%" width="100%"
  allowMultipleSelection="true"
  dataProvider="{stockDataAC}">
```

Selection can be carried out as shown in the following table.

Selection actions

Function	Action
Select	Click an item in the treemap.
Extend a selection	Hold down the CTRL key and click the items.
Reduce a selection	Hold down the CTRL key and click an already selected item to remove it from the selection.

Keyboard accessibility is also provided for interactions with treemap items. The keyboard actions for treemaps are shown in the following table.

Keyboard actions

Keyboard	Action
Right/left arrow keys	Navigate through the items that share a given parent item.
Up/down arrow keys	Navigate up and down the hierarchy.

For more information, see *Handling user interactions with charts* and *Selecting chart items* in the *Flex 3 Developer's Guide*.

Navigation

Once a single item has been selected, a typical action could be to drill down on that item. This can be done by setting the `virtualRoot` property of the `TreeMap` to the selected item as follows:

```
myTreeMap.virtualRoot = myTreeMap.selectedItems[0];
```

You can revert back to the treemap root by setting the `virtualRoot` property to `null`.

When you change the virtual root, a zooming effect can be performed. You can set its duration using the `animationDuration` style property. By default it is set to zero, which means that there is no effect.

For example, to get a 500ms animation use the following code.

```
<ilog:TreeMap width="200" height="400" animationDuration="500"/>
```

Customization

You can customize the color of the selection border by using the `selectionColor` style property of the `TreeMap`.

```
<ilog:Treemap height="100%" width="100%"
  selectionColor="cyan"
  dataProvider="{stockDataAC}">
```

Treemap legend control

You can add a treemap legend to your application to display which colors correspond to which values in the treemap. To do so, you define a `TreeMapLegend` control in MXML using the `<ilog:TreeMapLegend>` tag as follows.

```
<ilog:TreeMap width="200" height="400" id="myTreeMap"
                         dataProvider="{myDataProvider}" />
<ilog:TreeMapLegend width="200" height="50"
                         dataProvider="{myTreeMap}" />
```

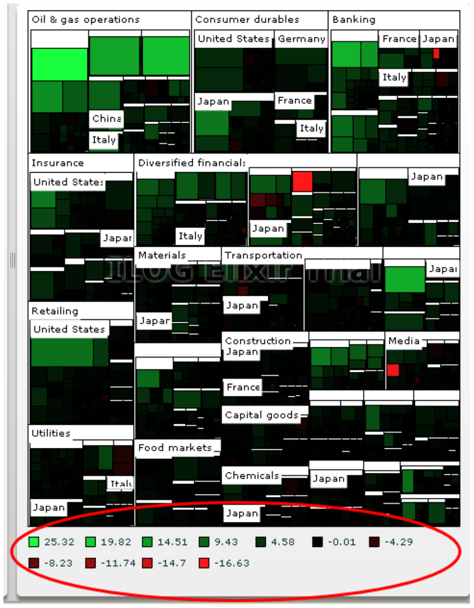
The `dataProvider` of the `TreeMapLegend` references the `TreeMap` control from which to display the legend data.

You can customize the following properties in a `TreeMapLegend`:

- The `direction` property to change the direction of the legend (horizontal or vertical)
- The color style property to change the text color of the legend
- The font style properties to change the text font of the legend
- The `labelFunction` property which provides a formatting function to compute the text of each legend based on the values in the treemap model

Example of treemap with legend

The following example shows a horizontal legend.



🟩 25.32 🟩 19.82 🟩 14.51 🟩 9.43 🟩 4.58 ■ -0.01 🟥 -4.29
🟥 -8.23 🟥 -11.74 🟥 -14.7 🟥 -16.63

Index

Symbols
, creating **249**

Numerics
100% property **42, 47**
3D
  composite chart **52**
3D chart
  architecture **33**
  controls **36**
  introduction **32**
  user interaction **63**

A
adding data to organization chart **579**
adding managed data fields
  organizational charts **579**
Adobe® Flex® **15**
Advanced Data Grid **155**
AdvancedDataGrid
  styles **190, 283**
AdvancedDataGrid class **215, 311**
  animations **211, 307**
algorithm property **636**
algorithms for treemaps **628**
allowMultipleSelection property **576, 644**
allowNavigation property **476**
allowSelection property **576**
alternating algorithm **628**
ambientLight property **56**
AmericasMap class **459**
angularAlternateFill property **601**
AngularAxis class **601, 605**
angularAxis property **605**
AngularAxisRenderer class **599, 601, 621**
angularAxisRenderers property **621**
angularFill property **601**
angularStroke property **601**
animate parameter **210, 306**
animate property **219, 316**
animation
  time-related **210, 306**
animationDuration property **480, 481, 644**
animations
  in resource chart **209**
  in task chart **305**
  on task expand/collapse **211, 307**
application file for printing **591**
architecture
  resource chart **157**
  task chart **247**
  task item renderer **181, 272**
AreaChart3D class **32, 47**
areaField field **633**
areaFill property **608, 622**
areaFill style property **608**
areaFunction property **635**
areas of same value **430, 447**
AreaSeries3D class **47**
ArrayCollection class **472, 632**
AsiaPacificMap class **459**
assistantField property **543**
associating tasks with resources **159**
axis
  radar chart renderers **621**
AxisRenderer3D class **60**

B
backgroundColor property **573**
backgroundFill tag **468**
bar renderer
  customizing the colors **379, 409**
  customizing the geometry **379, 409**
BarChart3D class **42**
BarSeries3D class **42**
BarSet3D class **42**

barycenter **484, 503**
`BitmapData` object **450**
BlackKnob **362**
blending of adjacent points **450**
`borderColor` property **637**
built-in animations
    in resource chart **209**
    in task chart **305**

C

calendar **242, 339**
    creating **69**
calendar classes **68**
calendars
    uses of **67**
`canDropLabel` property **601**
`Cartesian3D` class **57**
`CartesianChart3D` class **56, 60**
`CategoryAxis` class **32, 599, 605**
`categoryField` property **604**
`change` event **575, 642**
chart item renderers **622**
charts
    OLAP and pivot **508**
`children` property **167, 259, 542**
circle renderer **385**
`CircleItemRenderer` class **599, 614, 622**
`CircleRenderer` class **385**
circular gauges **344**
    building step by step **387**
    external assets **386**
    overview **368**
    visual elements **371**
circular radar charts **599**
`CircularBarRenderer` class **379**
`CircularGaugeAsset` class **386**
`CircularLabelRenderer` class **383**
`CircularScaleRenderer` class **373**
`CircularTrackRenderer` class **381**
class **136**
    `AngularAxis` **601, 605**
    `AngularAxisRenderer` **599, 601, 621**
    `AreaChart3D` **33, 47**
    `AreaSeries3D` **33, 47**
    `ArrayCollection` **472, 632**
    `AxisRenderer3D` **60**
    `AxisRendererChart3D` **33**
    `BarChart3D` **33, 42**
    `BarSeries3D` **33, 42**
    `BarSet3D` **42**
    `Cartesian3D` **57**
    `CartesianChart3D` **33, 56, 60**
    `CartesianSeries3D` **33**
    `CategoryAxis` **599, 601, 605**
    `CircleItemRenderer` **599, 614, 622**
    `CircleRenderer` **385**
    `CircularBarRenderer` **379**
    `CircularGaugeAsset` **386**
    `CircularLabelRenderer` **383**
    `CircularScaleRenderer` **373**
    `CircularTrackRenderer` **381**
    `ColumnChart` **603**
    `ColumnChart3D` **33, 37, 57**
    `ColumnSeries3D` **33, 37, 38, 42**
    `ColumnSet3D` **38**
    `ContinentalEuropeMap` **467**
    `CrossItemRenderer` **622**
    `dataProvider` **630**
    `DataTimeAxis` **605**
    `DateTimeAxis` **599**
    `DefaultDataDescriptor` **632, 633**
    `DiamondItemRenderer` **622**
    for a map **459**
    `LinearAxis` **605**
    `LineChart` **603**
    `LineChart3D` **33, 47**
    `LineSeries3D` **33, 45**
    `LogAxis` **599, 605**
    `MapBase` **476**
    `MapFeature` **460**
    `MapSymbol` **460**
    `MarkerRenderer` **407**
    `NeedleRenderer` **377**
    `NumericAxis` **605**
    `OrgChart` **539, 542, 543, 546, 557, 566, 573, 575, 576, 580**
    `OrgChartDataItem` **580**
    `OrgChartDetailedItemRenderer` **563**
    `orgChartDetailedItemRenderer` **539**
    `OrgChartEvent` **575**
    `orgChartEvents` **539**
    `OrgChartFields` **579**
    `OrgchartFields` **539**
    `OrgChartItem` **539, 566**
    `OrgChartItemRenderer` **562, 573**
    `orgChartItemRenderer` **539**
    `PieChart3D` **33, 56**
    `PieSeries3D` **33, 50**
    `PlotChart` **603**
    `PolarChart` **601**
    `RadarBubbleSeries` **601, 610**
    `RadarChart` **601, 605**
    `RadarColumnRenderer` **622**
    `RadarColumnSeries` **601, 617**
    `RadarGridLines` **599, 601**
    `RadarLineRenderer` **622**
    `RadarLineSeries` **601, 608, 622**
    `RadarPlotSeries` **601, 614**
    `RadarSeries` **601, 605**

RectangleRenderer **415**
RectangularBarRenderer **409**
RectangularGaugeAsset **416**
RectangularLabelRenderer **413**
RectangularScaleRenderer **403**
RectangularTrackRenderer **411**
Series3D **57**
SeriesInterpolate **64, 626**
SeriesSlide **64, 626**
SeriesZoom **64, 626**
SolidColor **480**
TreeMap **630, 632, 633, 636, 642, 645**
TreeMapLegend **646**
TriangleItemRenderer **599, 622**
UIComponent **460, 539, 566**
XML **632**
XMLList **632**
XMLListCollection **632**
class:
RadarBubbleSeries **622**
RadarColumnRenderer **622**
RadarColumnSeries **622**
RadarLineRenderer **622**
RadarLineSeries **622**
RadarPlotSeries **622**
classes
Gantt chart **157, 247**
heat map **432**
OLAP chart **511**
pivot chart **511**
resource chart **157**
task chart **247**
click in calendar sheet
events **108**
clipping a heat map **451**
clustered property **38, 42**
clustered radar column series **617**
collapse rows **215, 311**
collapsing tasks
with animation **211, 307**
collections of tasks **159, 250**
color model
for heat maps **430**
color model configuration **446**
color property **573**
color scheme, treemap **637**
colorEntry object **446**
colorField field **633**
colorField property **637**
colorFunction property **635, 637**
coloring a map **460**
ColorModel class **432**
colors
for heat maps **446**
colors of indicator characters **426**
colorScheme property **637**
ColumnChart class **603**
ColumnChart3D class **36, 37, 42, 57**
columnGroup property **617**
columns
hiding **166, 258**
columns in data grid **159**
ColumnSeries3D class **37, 38, 42**
ColumnSet3D class **38**
commit operation for task edits **330**
components **20**
composite 3D chart **52**
configuration
color model **446**
heat map data **442**
of valued heat map **447**
constraint data provider **250**
ContinentalEuropeMap class **459, 467**
converting coordinates **484**
coordinate conversion **484**
coordinates for maps **502**
creating a calendar **69**
creating controls
Treemap legend **646**
CrossItemRenderer class **622**
CSS classes
in resource chart **188**
in task chart **280**
current time in calendar
styling **101**
custom date format **237, 334**
custom functions
in recurrence descriptor **132**
custom functions for calendar data **75**
custom gauges **344, 365**
circular **367**
logical scales **369, 398**
rectangular **395**
custom map
barycenter file **505**
creating **495**
project file **505**
Custom Map Generator
advanced parameters **501**
main parameters **500**
mandatory parameters **500**
Custom Map Generator tool **494**
custom maps
barycenters in View pane **503**
interactions in View pane **503**
tooltips in View pane **503**
View pane **503**
customizing
predefined knobs **363**

customizing maps **465**
cylinder representation **39, 43**

D

data
    resource chart **159**
    task chart **250**
data clustering **628**
data events **217, 313**
data fields
    adding **579**
    in recurrence descriptor **132**
data for calendar **72**
data functions
    for heat maps **442**
data grid **155, 245**
    columns **159**
    event types **214, 310**
    styles **190, 283**
    user interaction **222, 320**
data in calendar
    events **104**
data provider
    constraint **250**
    for heat maps **430, 442**
    resource **159**
    task **159, 250**
data providers **155, 158, 159, 245, 249, 250**
data providers for organization charts **542**
data providers for treemaps **632**
data tip **114, 115, 172, 264**
`dataDescriptor` property **632**
`dataField` property **604**
`dataProvider` property **37, 42, 45, 47, 50, 54, 472, 540, 542, 544, 546, 554, 558, 559, 628, 630, 632, 633, 637, 646**
`DataTimeAxis` class **605**
`dataTipField` property **172**
`dataTipFunction` property **172**
date format for recurrence **137**
date in calendar **80**
dates **237, 334**
`DateTimeAxis` class **32, 599**
default item renderers
    subclassing **581**
default task item renderer **180, 270**
default tooltip **172, 264**
`DefaultDataDescriptor` class **632, 633**
`DensityHeatMap` class **430, 432, 436, 450**
deploying an application **27**
depth parameter **32**
`DiamondItemRenderer` class **622**
direct manipulation **227, 325**
direction of legend in treemap **646**
dragging a task **230, 327**
dragging in Gantt sheet **224, 323**
dragging tasks **224, 323**
drawing **484**
`drawingCanvas` property **484**
drilldown action **644**

E

E4X **19, 72**
editing data values **344**
editing tip **114, 172, 264**
`editKind` property **235, 332**
effects (radar chart) **626**
elevation parameter **32**
`elevationAngle` property **56**
end of an edit session **236, 333**
end of recurrence **136**
`EuropeMiddleEastAfricaMap` class **459**
event **230, 327**
    `change` **575, 642**
    `itemClick` **575, 642**
    `itemDoubleClick` **575, 642**
    `itemRollOut` **575, 642**
    `itemRollOver` **575, 642**
    `mapChange` **474**
    `mapItemDoubleClick` **474**
    `mapItemRollOut` **474**
    `mapItemRollOver` **474**
    `maptemClick` **474**
    recurring **130**
event data tool tips **114**
event in calendar
    long **90**
    short **88, 92**
event listener **231, 234, 235, 328, 331, 332**
event notification in calendar **109**
events **72, 155, 245**
    in resource charts **213, 309**
    navigating in resource chart **219**
    navigating in task chart **316**
    resource selection **214, 310**
    user interactions **217, 313**
events (radar chart) **624**
events for treemaps **642**
events in calendar **103**
events on tasks **224, 227, 323, 325**
exception dates **147**
exceptions to recurrence rule **147**
expand rows **215, 311**
expanding tasks
    with animation **211, 307**
external assets for gauges **386, 416**

F

field mappings
    for heat maps **442**
field mappings for calendar data **73**
fields

areaField **633**
colorField **633**
labelField **633**
file, Shock Wave **27**
fill property **57, 466**
fillColor property **637**
fillFunction property **57**
fills property **57**
floor of date **242, 339**
focus on time **223, 322**
form property **45**
formatting scale labels **373, 403**
formatting scale tick marks **373, 403**
FormPrintView.mxml file **591**
frequency of recurrence **136**
full circular gauges **355**

G

Gantt chart **155, 245**
Gantt sheet **155, 245**
interaction **224, 323**
styles **191, 284**
time-related animation **210, 306**
Gantt sheet events **219, 316**
GanttDataGrid class **157**
GanttSheet class **157, 217, 235, 313, 332**
GanttSheetEvent class **157, 217, 236, 313, 333**
GanttSheetEventReason class **236, 333**
gap between indicator character characters **426**
gap between indicator characters **426**
gauges
custom **344, 365**
external assets **386, 416**
framework **366**
logical scales **369, 398**
predefined **344, 354**
value renderer classes **376, 406**
gauges architecture
framework classes **348**
predefined circular gauges classes **348**
predefined rectangular gauges classes **348**
getFeature() method **460**
global view mode **537, 558**
grids
in Gantt sheet **191, 284**
grids in calendar
styling **96**
grouping columns **617**

H

headers in calendar
styling **99**
heat map
on a map **451**
with legend **454**
heat map classes **432**
heat map data configuration **442**
heat maps **430**
HeatMapBase class **432**
height of indicator characters **426**
hierarchical objects **633**
HierarchicalData class **15**
highlight interval **223, 322**
horizontal gauges **344, 358**
horizontalAxis property **57**
horizontalAxisStyleName property **60**

I

iCalendar specification **130, 137**
iCalendar standard **136**
ICollectionView interface **632**
iconFunction property **167, 259**
icons for resources **167, 259**
identifiers
of tasks and resources **159**
IHierarchicalCollectionView interface **542**
IHierarchicalData interface **542**
IListItemRenderer interface **566**
ilog:TreeMapLegend tag **646**
indicator
14–segment alphanumeric **424**
35–segment alphanumeric **424**
numeric **424**
indicator character segments
gap between **426**
indicator characters
colors **426**
gap between **426**
height **426**
indicators **422**
initialPosition property **558**
instance **242, 339**
interaction with calendar **110**
interaction with resource chart **217**
interaction with task chart **313**
interactions
radar chart **625**
interactions in View pane **503**
interactions, organization chart **576**
ISO8601:2000 date format **237, 334**
isometric contours **430, 447**
item renderer fields, organization chart **543**
item renderers (radar chart) **622**
item renderers (radar charts) **622**
item renderers in calendar **87**
itemClick event **62, 575, 576, 624, 642, 644**
itemClickAction function **62, 624**
itemDoubleClick event **575, 642**
itemEditEnd event **237, 334**
itemEditMove event **237, 334**
itemEditResize event **237, 334**

`itemRenderer` property **234, 331, 614**
`itemRollOut` event **575, 642**
`itemRollOver` event **575, 642**
items in calendar **87**
    selecting **110**
    styling **100**
`itemStyleName` property **573**

K

kind of editing **235, 332**
knobs **344**
    predefined **362**

L

label renderer
    custom **383, 413**
    rendering text **383, 413**
`labelField` field **633**
`labelFunction` property **635**
`labelPosition` property **50**
`labelThreshold` property **636**
latitude/longitude **484**
`layout` field **550, 551, 552, 553, 555**
`layoutField` property **543**
`layoutXPadding` property **570**
`layoutYPadding` property **570**
left page in printing **591**
legend
    for heat map **454**
legend on treemap **646**
light direction parameter **32**
`lightLatitude` property **56**
`lightLongitude` property **56**
`LinearAxis` class **605**
`LineChart` class **603**
`LineChart3D` class **47**
`LineSeries3D` class **45**
`linkColor` property **572**
`linkThickness` property **572**
list-based data provider **159, 250**
local view mode **537, 559**
localItem property **559**
localizing an application **28**
`LogAxis` class **32, 599, 605**
logical scales **369, 398**
long event in calendar **90**
`lowerLevelLimit` property **559**

M

managed data fields **579**
mandatory custom map parameters **500**
map
    with heat map **451**
map control **463**
map coordinates **502**
map customization **465**
`MapBase` class **459, 476**
`MapBase.FILL` constant **480**
`MapBase.HIGHLIGHT_FILL` constant **480**
`MapBase.HIGHLIGHT_STROKE` constant **480**
`MapBase.STROKE` constant **480**
`mapChange` event **474**
`MapEvent` class **474**
`MapEvent` event **474**
`MapFeature` class **460**
`MapHeatMap` class **430, 432, 438, 451**
`mapItemClick` event **474**
`mapItemDoubleClick` event **474**
`mapItemRollOut` event **474**
`mapItemRollOver` event **474**
mapping a field to a property **633**
mapping a field to a property/attribute **544**
`MapSymbol` class **460**
marker renderer
    customizing the geometry **407**
`MarkerRenderer` class **407**
`maxBottomMargin` property **636**
maximum visible time **170, 262**
`maxLeftMargin` property **636**
`maxRadius` property **610**
`maxRightMargin` property **636**
`maxTopMargin` property **636**
meeting in calendar **139**
messages **28**
method
    `centreOn` **576**
    `fitToContents` **576**
    `fitToSubTree` **576**
    `scale` **576**
    `zoomBy` **576**
milestone **155, 161, 180, 245**
`minField` property **38, 42**
minimum duration for resizing **237, 334**
minimum visible time **170, 262**
modifying a recurring event **146**
mouse actions **223, 322**
    Gantt sheet **224, 323**
`moveEnabled` property **229, 326**
moving tasks
    snapping **177, 268**
multipage printing **586**
`mx.charts.renderers` package **614**

N

navigable time range **170, 262**
navigating items (radar chart) **625**
navigation **155, 245**
    with animation **210, 306**
    with mouse **223, 322**
navigation in calendar
    events **105**

needle renderer
    customizing the geometry **377**
NeedleRenderer class **377**
neutral property **637**
none property **50**
nonworking periods **176, 267**
nonworking periods in calendar **84**
nonWorkingDays property **176**
nonWorkingRanges property **176**
notification of events in calendar **109**
NumericAxis class **605**

O

object **149**
objects **149, 242, 339**
oblique projection **32**
OLAP chart
    creating **514**
OLAP charts **508**
OLAP cube **514**
OLAP query **514**
OLAP schema **514**
organization chart
    adding managed data fields **579**
    printing **585**
organization charts
    introduction **537**
OrgChart class **539, 542, 543, 546, 557, 566, 573, 575, 576**
    subclassing **580**
OrgChartDataItem class
    subclassing **580**
OrgChartDetailedItemRenderer class **539, 563**
OrgChartEvent class **575**
OrgChartEvents class **539**
OrgChartField class
    subclassing **579**
OrgChartFields class **539**
OrgChartItem class **539, 566**
OrgChartItemRenderer class **539, 562, 573**
orthographic projection **32**
outside property **50**
overlaid property **38, 42, 47**

P

package
    mx.charts.renderers **614**
page and label controls **591**
page breaks **588**
palette
    for density heat map **450**
PieChart class **36**
PieChart3D class **32, 36, 56**
PieSeries3D class **50**
pivot charts **508**
PlotChart class **603**
PolarChart class **601**
polygonal radar charts **599**
porperty
    taskStartTimeFunction **237**
predefined circular gauges
    customizing **356**
    full circular **355**
    semicircular **355**
predefined gauges **344**
predefined knobs **362**
    customizing **363**
predefined rectangular gauges
    customizing **360**
    horizontal **358**
    vertical **358**
Print button **591**
print layout properties **588**
printing
    organization chart **585**
printingScalingFactor property **590**
printingScalingMode property **590**
project file **505**
project, setting up **24**
projectionType property **56**
property **113, 576**
    algorithm **636**
    allowMultipleSelection **644**
    angularAlternateFill **601**
    angularAxis **605**
    angularAxisRenderers **621**
    angularFill **601**
    angularStroke **601**
    animationDuration **644**
    areaFill **608, 622**
    areaFunction **635**
    backgroundColor **573**
    borderColor **637**
    canDropLabel **601**
    categoryField **604**
    children **167, 259**
    color **573**
    colorField **637**
    colorFunction **635, 637**
    colorScheme **637**
    columnGroup **617**
    dataDescriptor **632**
    dataField **604**
    dataProvider **628, 632, 633, 637, 646**
    fillColor **637**
    iconFunction **167, 259**
    itemRenderer **614**
    labelFunction **635**
    labelThreshold **636**
    maxBottomMargin **636**

maxLeftMargin **636**
maxRadius **610**
maxRightMargin **636**
maxTopMargin **636**
moveEnabled **229, 326**
neutral **637**
nonWorkingDays **176**
nonWorkingRanges **176**
radialAlternateFill **601**
radialAxis **605**
radialAxisRenderers **621**
radialFill **601**
radialStroke **601**
radiusField **610**
reason **236, 333**
renderFunction **637**
resizeEnabled **229, 326**
resourceDataProvider **159**
rollOverBackgroundColor **573**
rollOverBorderColor **573**
rollOverTextColor **573**
selectedBackgroundColor **573**
selectedBorderColor **573**
selectedRollOverBackgroundColor **573**
selectedRollOverBorderColor **573**
selectedRollOverTextColor **573**
selectedTextColor **573**
selectionColor **645**
selectionMode **625**
snappingTimePrecision **177, 268**
stroke **601**
taskDataProvider **159**
taskEndTimeFunction **237, 334**
taskItemRenderer **229, 326**
taskStartTimeFunction **334**
tickLength **601**
tickPlacement **601**
topMarginProportion **636**
virtualRoot **644**
visibleTimeRangeEnd **169, 261**
visibleTimeRangeStart **169, 261**
WorkingTimes **176**
property (style)
areaFill **608**

R

radar chart
creating **603**
data **604**
RadarBubbleSeries class **601, 610, 622**
RadarChart class **601, 605**
RadarColumnRenderer class **622**
RadarColumnSeries class **601, 617, 622**
RadarGridLines class **599, 601**
RadarLineRenderer class **622**
RadarLineSeries class **601, 608, 622**
RadarPlotSeries class **601, 614, 622**
RadarSeries class **601, 605**
radial gradient **450**
radialAlternateFill property **601**
radialAxis property **605**
radialAxisRenderers property **621**
radialFill property **601**
radialStroke property **601**
radiusField property **610**
reason property **236, 333**
reassignEnabled property **229**
rectangle renderer **415**
RectangleRenderer class **415**
rectangular gauges
building step by step **417**
external assets **416**
overview **396**
visual elements **401**
RectangularBarRenderer class **409**
RectangularGaugeAsset class **416**
RectangularLabelRenderer class **413**
RectangularScaleRenderer class **403**
RectangularTrackRenderer class **411**
recurrence classes **131**
recurrence descriptor
custom functions **132**
data fields **132**
recurrence engine **130**
recurrence example
birthday **139**
meeting **139**
recurrence parameters **136**
recurrence rule **130**
exceptions **147**
recurrence rule class **136**
recurrence rule specification **137**
recurrence set **147**
recurring event
modifying **146**
preventing editing **148**
recurring event in calendar **130**
reminder of events in calendar **109**
renderFunction property **637**
rendering, treemap examples **639**
Repeater class **472**
resizeEnabled property **229, 326**
resizing tasks
snapping **177, 268**
resource bundles **28**
resource chart **155**
interaction events **217**
styles **188**
resource columns

hiding **166**
resource data provider **159**
resource icons **167, 259**
resource selection events **214, 310**
ResourceChart class **157**
ResourceChart, creating **158**
resourceDataProvider property **159**
reverseStep property **45**
RFC-2445 **130**
Rich Desktop Applications **15**
Rich Internet Applications **15**
rollOverBackgroundColor property **573**
rollOverBorderColor property **573**
rollOverTextColor property **573**
rotation parameter **32**
rotationAngle property **56**
row grid **155, 245**
RSL application **27**

S

scale labels **373, 403**
scale renderer
custom **373**
formatting labels **373, 403**
formatting tick marks **373, 403**
scale tick marks **373, 403**
scheduling resources **155**
scheduling tasks **245**
scroll time **223, 322**
segment property **45**
selectedBackgroundColor property **573**
selectedBorderColor property **573**
selectedItems property **576**
selectedRollOverBackgroundColor property **573**
selectedRollOverBorderColor property **573**
selectedRollOverTextColor property **573**
selectedTextColor property **573**
selecting items (radar chart) **625**
selecting items in calendar **110**
selecting tasks **224, 323**
selecting time range in calendar **110**
selecting treemap items **644**
selectionColor property **645**
selectionMode property **63, 476, 625**
semicircular gauges **355**
series types (radar) **607**
Series3D class **57**
SeriesInterpolate class **64, 626**
SeriesSlide class **64, 626**
SeriesZoom class **64, 626**
setStyle() method **480**
setting up a project **24**
shapefile **495**
Shock Wave file **27**
short event in calendar **88, 92**
showLabel() function **591**
SimpleKnob **362**
size of printed organization chart **590**
slice-and-dice algorithm **628**
snap task when moving **237, 334**
snapping tasks **177, 268**
snappingTimePrecision property **177, 237, 268, 334**
SolidColor class **480**
specification of recurrence rule **137**
squarified algorithm **628**
stacked property **42, 47**
startTime and endTime of task **234, 331**
step property **45**
stroke property **57, 466, 467, 469, 601**
styleName property **573**
styles
for data grid **190, 283**
for Gantt sheet **191, 284**
for Gantt sheets **198, 291**
for tasks **204, 297**
styles for resource charts **188**
styles for task charts **280**
summary data **514**
system requirements **15**

T

task
tooltip **172, 264**
task chart **245**
interaction events **313**
styles **280**
task columns
hiding **258**
task data items **159**
task data provider **159, 181, 250, 272**
task editing sequence **230, 327**
task events in Gantt sheet **224, 323**
task item renderer
architecture **181, 272**
default **180, 270**
taskDataProvider property **159**
taskEndTimeFunction property **237, 334**
TaskItem class **157**
TaskItem object **181, 272**
TaskItemRenderer class **157, 204, 297**
taskItemRenderer property **229, 326**
tasks **155**
leaf **245**
styles **204, 297**
summary **245**
taskStartTimeFunction property **237, 334**
tickLength property **601**
tickPlacement property **601**
time axis **171, 263**
time grid **155, 245**

time indicator **155, 245**
time range
    change **219, 316**
time range in calendar **106**
    selecting **110**
time range in resource chart
    visible **169**
time range in task chart
    visible **261**
time range selection tool tips **115**
time scale **155, 245**
    styles **198, 291**
    user interaction **223, 322**
time units **242, 339**
time-related animation **210, 306**
`TimeScale` class **157, 198, 291**
tool tip
    defining **114**
    event data **114**
    time range selection **115**
tooltips **172, 264**
tooltips in View pane **503**
top page in printing **591**
`topMarginProportion` property **636**
track renderer
    customizing the colors **381, 411**
    customizing the geometry **381, 411**
`TreeMap` class **630, 632, 633, 636, 642, 645**
treemap legend
    creating a control **646**
`TreeMapEvent` event **642**
`TreeMapLegend` class **646**
`TriangleItemRenderer` class **599, 622**
`type` property **37, 47**

U

`UIComponent` class **460, 470, 477, 539, 566**
`upperLevelLimit` property **559**
user interaction
    Gantt sheet **224, 323**
    time scale **223, 322**
user interactions
    data grid **222, 320**
user interactions with resource charts **221**
user interactions with task charts **319**
`USStatesMap` class **459**
utility classes **149**

V

value renderer classes **376, 406**
valued heat maps **430**
`ValuedHeatMap` class **430, 432, 434, 447**
vertical gauges **344, 358**
`verticalAxis` property **57**
`verticalAxisStyleName` property **60**
view mode in calendar **80**
view modes **557**
`virtualRoot` property **644**
`visible` property **166, 258**
visible time
    min and max **170, 262**
visible time range **169, 261**
visible time range in calendar **80**
`visibleTimeRangeChange` event **219, 316**
`visibleTimeRangeEnd` property **169, 261**
`visibleTimeRangeStart` property **169, 261**
visual elements
    circular gauges **371**
    `CircularBarRenderer` **379**
    `CircularLabelRenderer` **383**
    `CircularScaleRenderer` **373**
    `CircularTrackRenderer` **381**
    `MarkerRenderer` **407**
    `NeedleRenderer` **377**
    rectangular gauges **401**
    `RectangularBarRenderer` **409**
    `RectangularLabelRenderer` **413**
    `RectangularScaleRenderer` **403**
    `RectangularTrackRenderer` **411**

W

work grid **155, 176, 245, 267**
working periods **176, 267**
working periods in calendar **84**
`WorkingTimes` property **176**
`WorldCountriesMap` class **459**

X

`XML` class **632**
`XMLList` class **632**
`XMLListCollection` class **632**

Z

zoom factors
    min and max **171, 263**
zoom on time **223, 322**
`zoomableSymbols` property **470**
`zoomBy` method **576**